VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2010 **Bc. Petr Havíček**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy pro směrování dopravních vozidel
Algorithms for Routing Transport Vehicles

2010 **Bc. Petr Havíček**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Havíček**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Algoritmy pro směrování dopravních vozidel
Algorithms for Routing Transport Vehicles

Zásady pro vypracování:

V rámci diplomové práce se bude diplomant zabývat problematikou vyhledávání nejvhodnějších cest pro směrování (routing) pro navigaci dopravního vozidla. Diplomant obdrží reálná data dopravní sítě české republiky.

Jednotlivé body zadání:
1. Diplomant prostuduje problematiku směrování.
2. Diplomant implementuje vybrané algoritmy pro směrování.
3. Implementované směrovací algoritmy budou doplněny i o různá omezení v rámci dopravní situace a vozidel (směrování pro kamionovou dopravu, směrování pro osobní dopravu).
4. Na různých experimentech vyhodnotí vhodnost vybraných algoritmů pro směrování.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 20.11.2009
Datum odevzdání: 07.05.2010

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 1.5.2010 _____

Poděkování

Na tomto místě bych rád poděkoval Ing. Janu Martinovičovi, Ph.D., za cenné rady, připomínky, ochotu a čas, který mi věnoval při tvorbě této práce.

Abstrakt a klíčová slova

Abstrakt

Tato práce se zabývá problematikou směrování nejen dopravních vozidel. Jsou v ní uvedeny algoritmy, které se používají k tomuto účelu. Tyto algoritmy hledají nejkratší nebo nejrychlejší cesty a cesty s různým dopravním omezením. Všechny uvedené algoritmy, jsou popsány, je uveden princip jejich funkce a jejich výhody a nevýhody.

Vzhledem k tomu, že jsme měli k dispozici data silniční sítě České republiky, tak bylo možno tyto algoritmy otestovat při hledání cest nad reálnými daty.

Díky několika provedeným optimalizacím jsme schopni vyhledat cesty stejné nebo podobné cesty v stejném čase jako ostatní aplikace zabývají se touto problematikou.

Klíčová slova

Směrování vozidel, Dijkstrův algoritmus, A* algoritmus, optimalizace směrovacích algoritmů

Abstract

This work deals not only with the routing of transport vehicles. In this work, given algorithms that are used for this purpose. These algorithms find the shortest or fastest routes and routes with different traffic restrictions. All the algorithms are described, is the principle in their work and their advantages and disadvantages.

Given that we had available data of the road network of the Czech Republic, it was thus possible to test these algorithms in finding ways to real data.

Thanks to some optimizations, that we made, we can find ways identical or similar trips in the same time as other applications dealing with this issue.

Key words

Vehicle routing, Dijkstra's algorithm, A * algorithm, Optimization of routing algorithms

Seznam použitých symbolů a zkratek

GPS (Global Positioning System) – Je systém, s jehož pomocí je možno určit polohu kdekoliv na Zemi s přesností první desítky metrů.

MB – Jednotka kapacity paměti počítače, rovná se 1 048 576 bytů.

Obsah

Seznam použitých obrázků

Seznam použitých tabulek

1 Úvod

V této práci se budeme zabývat algoritmy pro směrování vozidel a jejich praktickou aplikací na silniční síť České republiky. Hlavním důvodem vzniku této práce je požadavek společnosti AXITECH na vytvoření aplikace pro směrování dopravních vozidel. Z tohoto důvodu se tato práce nezabývá pouze teoretickými možnostmi, ale jde hlavně o to, vytvořit fungující aplikaci pro směrování vozidel, která je srovnatelná s ostatními již vytvořenými aplikacemi tohoto druhu.

Pod pojmem směrování si můžeme představit hledání cesty mezi dvěma zadanými místy. Cesta, která se má vyhledat může být nejrychlejší nebo nejkratší, případně s dalšími omezeními jako například poplatky, či omezeními pro nákladní dopravu.

Než jsme mohli vůbec začít s vytvářením požadované aplikace, bylo zapotřebí se seznámit s již existujícími aplikacemi pro směrování vozidel. Toto seznámení je uvedeno v druhé kapitole, kde se nachází rozbor jednotlivých webových aplikací a porovnání výsledků vyhledávání mezi nimi.

Dalším úkolem bylo seznámení se s algoritmy, které jsou vhodné pro směrování vozidel. Mezi tyto algoritmy patří Dijkstrův algoritmus a A* algoritmus. Teoretickým popsáním a vysvětlení funkce těchto algoritmů se zbývá třetí kapitola.

Kapitola čtyři a pět se zabývá popisem implementace obou zmíněných algoritmů. Jsou v ní také uvedeny jednotlivé třídy, které využívají jednotlivé algoritmy.

Nejdůležitější část této práce se nachází v šesté kapitole. V ní jsou uvedeny optimalizace a experimenty jednotlivých algoritmů, porovnání algoritmů mezi sebou a porovnání výsledků algoritmů s výsledky z již existujících aplikací pro hledání cest.

1.1 Struktura práce

V kapitole 2 jsou prozkoumány některé webové aplikace pro směrování vozidel a jsou porovnány mezi sebou. Kapitola 3 vysvětluje princip algoritmů pro směrování a popisuje algoritmy k tomu určené. Mezi tyto algoritmy patří Dijkstrův algoritmus (sekce 3.1) a A* algoritmus (sekce 3.2). Kapitola 4 se zabývá implementací Dijkstrova algoritmu a kapitola 5 se zabývá implementací A* algoritmu.

Nejdůležitější část této práce jsou optimalizace a experimenty, ty jsou uvedeny v kapitole 6. V sekci 6.1 jsou uvedeny optimalizace zdroje dat. Optimalizace Dijkstrova algoritmu jsou uvedeny v sekci 6.2 a optimalizace A* algoritmu jsou uvedeny v sekci 6.3. Zhodnocení výsledků optimalizací je uvedeno v kapitole 6.4.

Kapitola 7 se zabývá principem vykreslování nalezené cesty. Celkové zhodnocení výsledků práce je uvedeno v závěru, jenž se nachází v kapitole 8.

2 Webové aplikace pro hledání cest

V současné době existuje celá řada aplikací pro vyhledávání cest. Tyto aplikace se dají rozdělit na dvě hlavní skupiny. Do první skupiny patří aplikace, které pracují na mobilních zařízeních a jsou umístěny ve vozidle, kde směrují podle aktuální pozice. Ve druhé skupině jsou aplikace, které cestu najdou, ale výsledek je potřeba někam uložit či vytisknout, abychom ho posléze mohli využít.

Hlavně díky zvyšování výkonu a paměti v mobilních zařízeních jsou dnes poměrně rozšířené navigace do aut. Tyto navigace pracují v reálném čase a podle GPS (Global Positioning System) polohy, kde se právě nacházíte, určují cestu do požadovaného cíle. U takovéhoto typu aplikací je potřeba občasná aktualizace map z důvodu přidání nových cest, případně jiných důležitých věcí.

Pokud potřebujeme najít cestu jenom občas anebo nechceme kupovat navigaci do auta, tak nám dobře poslouží webové aplikace na hledání cest. Mezi nejznámější patří mapy.cz a maps.google.cz. Jejich hlavní výhoda spočívá v tom, že jsou zdarma a stále aktuální.

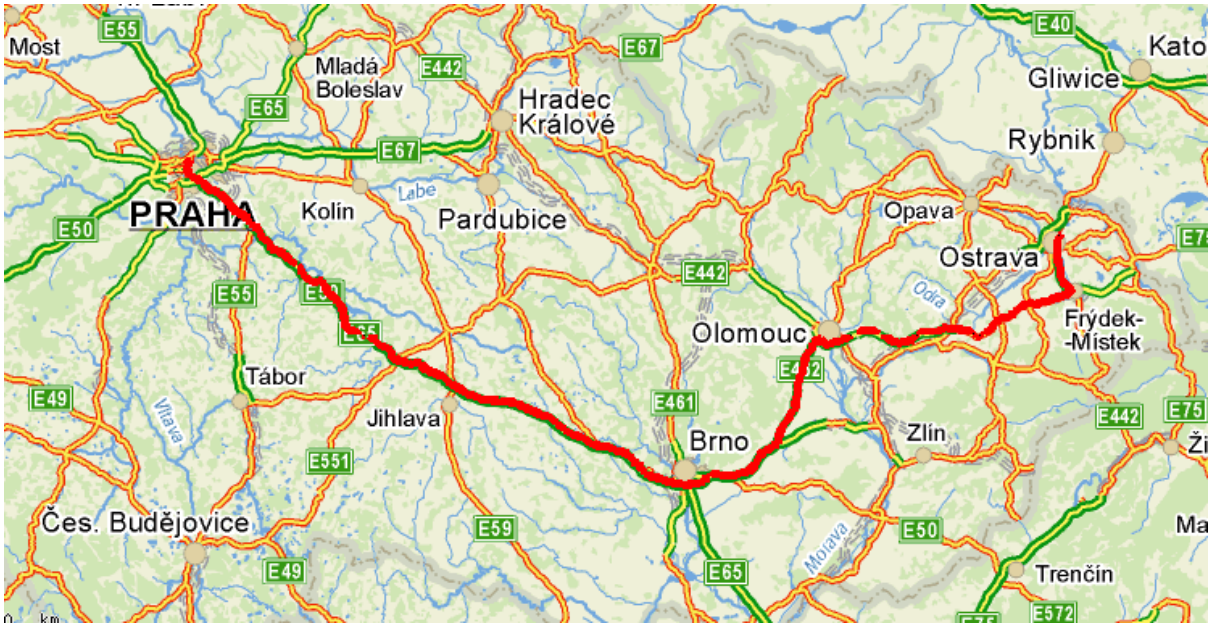
Tyto webové aplikace budou složit pro porovnání výsledků hledání mezi sebou a dále pro porovnání výsledků naimplementovaných algoritmů. Z toho důvodu budou v následující kapitole popsány jejich vlastnosti a bude ukázán výsledek hledání nejrychlejší cesty z Prahy do Ostravy. Aby bylo možno výsledky porovnat co nejlépe, bude startovní a cílové místo zadáno pomocí GPS souřadnic a to Praha 50°5′16.22″N 14°25′26.87″E a Ostrava 49°50′4.73″N 18°16′55.36″E.

2.1 Mapy.cz

Ve většině případů, pokud budeme chtít vyhledat cestu, tak nás bude zajímat nejrychlejší cesta. Nicméně někdy nás bude zajímat i nejkratší cesta. Jak vidíme na obrázku 2-1, tak mapy.cz umí vyhledávat oba typy cest, nejkratší i nejrychlejší. Další důležitou funkcí je možnost vybrat zda se mají hledat placené či neplacené cesty. Tato funkce je taktéž obsažena.

Obrázek 2-1: Možnosti hledání cest a zobrazení výsledku hledání

Výsledná nejrychlejší cesta z Prahy do Ostravy je dlouhá 391,4 km a její výsledný čas je 3 hodiny a 38 minut. Cesta je zobrazena na obrázku 2-2. Jak je vidět, tak je nalezená cesta vedena z Prahy po dálnici do Brna, dále pak přes Olomouc, Frýdek-Místek a Ostrava.



Obrázek 2-2: Zobrazení nejrychlejší cesty Praha – Ostrava pomocí aplikace Mapy.cz

2.2 Maps.google.cz

Hledání cesty na maps.google.cz umožňuje najít pouze nejrychlejší cestu, nejkratší cesty hledat neumí. Další funkce, která je podporována, je hledání včetně placených či neplacených úseků. Na obrázku číslo 2-3 je zobrazení nalezené nejrychlejší cesty Praha – Ostrava, jejíž délka je 384 km a doba trvání je 4 hodiny 6 minut.

Obrázek 2-3: Zobrazení nejrychlejší cesty Praha – Ostrava pomocí aplikace Maps.google.cz

2.3 Amapy.centrum.cz

Další webovou aplikací na hledání cesty, kterou jsem vybral na porovnání je amapy.centrum.cz. Tato aplikace podporuje vyhledávání jak nejrychlejších, tak i nejkratších cest. Samozřejmostí je také zohledňovat placené či neplacené úseky při hledání cesty. Nejrychlejší cesta je zobrazena na obrázku 2-4. Ta je dlouhá 394,75 km a trvá 4 hodiny a 2 minuty.

Obrázek 2-4: Zobrazení nejrychlejší cesty Praha – Ostrava pomocí aplikace Amapy.centrum.cz

2.4 Porovnání aplikací

Pokud se podíváme na předchozí příklady hledání nejrychlejší cesty z Prahy do Ostravy, tak vidíme, že se nalezené cesty od sebe liší, jak vzdáleností, tak i dobou trvání jízdy. Souhrnný přehled nalezené nejrychlejší cesty je uveden v tabulce 2-1.

Nejrychlejší cestu s poplatky našla webová aplikace mapy.cz, jejíž čas je 3 hodiny a 38 minut a délka cesty je 391,4 km. Zbývající dvě aplikace našli nejrychlejší cestu s podobným časem a to 4 hodiny a 6 minut maps.google.cz a 4 hodiny a 2 minuty amapy.centrum.cz. Nicméně je potřeba podotknout, že přestože maps.google.cz našla cestu, která v porovnání s ostatními uvedenými trvá nejdéle, tak je nalezená cesta nejkratší. Nalezené délky cest jsou v rozsahu deseti kilometrů 384 km (maps.google.cz) – 394,74 km (amapy.centrum.cz).

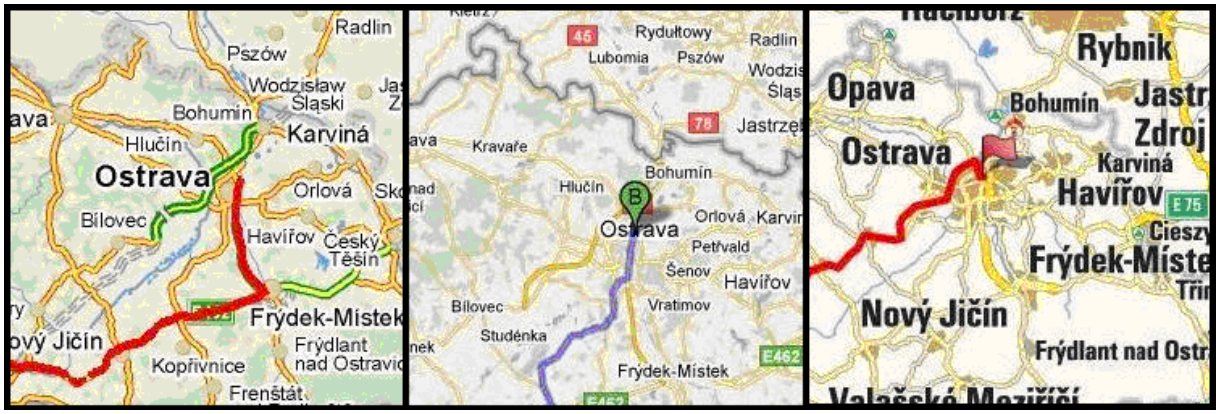
Pokud se zaměříme na výsledné časy, pak je vidět že čas vypočtený mapy.cz je skoro o 25 minut kratší než u zbylých dvou, přitom se délka cesty liší jen o pár kilometrů. Díky tomu můžeme předpokládat, že při výpočtu času jízdy používá každá aplikace vlastní způsob výpočtu.

Nejrychlejší	Poplatky		Bez poplatků	
Aplikace	Délka (km)	Čas	Délka (km)	Čas
mapy.cz	391,4	3h 38m	336.9	4h 44m
maps.google.cz	384	4h 6m	339	5h 57m
amapy.centrum.cz	394,74	4h 2m	341.51	4h 33m

Tabulka 2-1: Porovnání hledání nejrychlejší cesty s poplatky a bez poplatků

Při hledání nejrychlejší cesty bez poplatků jsou nalezeny cesty v rozsahu pěti kilometrů. Nicméně pokud se podíváme na výsledný čas cesty, výsledky jsou velice zajímavé. Mapy.cz našly cestu časové délky 4 hodiny a 44 minut a amapy.cz našly cestu trvající 4 hodiny 33 minuty. Rozdíl mezi těmito cestami je přibližně 10 minut, což není nějak hrozné. Pokud ale porovnáme tyto dva časy s časem cesty od maps.google.cz pak čas o 73 minut horší oproti mapy.cz a dokonce o 84 minut horší než amapy.centrum.cz. Je otázkou, který z těchto časů se co nejvíce blíží skutečné hodnotě. Každopádně získat rozdíl časů blížící se hodině a půl je poměrně neobvyklé.

Na obrázku 2-5 vidíme detail příjezdu do Ostravy pro nejrychlejší cestu s poplatky. Každá z aplikací našla jiný příjezd do Ostravy. Mapy.cz preferují nejrychlejší cestu přes Frýdek-Místek. Zatímco amapy.centrum.cz cestu přes Bílovec. Rozdíl v hledaných cestách může být způsoben algoritmem hledání cesty, každá aplikace může používat jiné algoritmy pro hledání cest a výpočtů doby trvání cesty.



Obrázek 2-5: Příjezd do Ostravy: mapy.cz, maps.google.cz a amapy.centrum.cz

Výsledky hledání nejkratších cest mezi Prahou a Ostravou jsou uvedeny v tabulce 2-2. Je celkem zajímavé, že na maps.google.cz není možnost vyhledat nejkratší cestu. Nejkratší cesta byla nalezena aplikací amapy.centrum.cz o délce 325,7 metrů a celkový čas jízdy trvá 5 hodin a 9 minut. Mapy.cz nalezly cestu o délce 328.1 km, což je o 2,5 km více než předchozí výsledek. Časy jízdy obou cest jsou téměř totožné. Na nejkratší cestě mezi Prahou a Ostravou se nenacházejí žádné placené úseky, to je vidět z porovnání výsledků hledání cest s poplatky a bez nich. Jsou totiž totožné.

Nejkratší	Poplatky		Bez poplatků	
Aplikace	Délka (km)	Čas	Délka (km)	Čas
mapy.cz	328.1	5h 11m	328.1	5h 11m
maps.google.cz	nepodporováno		nepodporováno	
amapy.centrum.cz	325.7	5h 9m	325.7	5h 9m

Tabulka 2-2: Porovnání hledání nejkratší cesty s poplatky a bez poplatků

Abych docílil co možná největší přesnosti při vyhledávání, byly souřadnice zadané pomocí GPS polohy, které by pro Prahu 50°5′16.22″N 14°25′26.87″E a pro Ostravu 49°50′4.73″N 18°16′55.36″E. Tímto způsobem jsou eliminovány chyby v přesnosti vzniklé důsledkem nepřesného zadání počátečního a koncového místa. Tudíž lze dané výsledky považovat za maximálně přesné ukázání, jak umí daná webová aplikace vyhledávat požadované cesty.

Úkolem ukázaných příkladů na vyhledání nejkratší a nejrychlejší cesty mezi Prahou a Ostravou, je ukázat jak jsou realizovány současné aplikace na vyhledávání cest a porovnat jejich výsledky mezi sebou. Díky tomu bylo zjištěno několik zajímavých faktů.

Asi největší shoda nastala v případě, kdy jsme hledali nejkratší cestu ať už s poplatky či bez nich. To je možná způsobeno i tím, že maps.google.cz nepodporuje toto vyhledávání, tudíž můžeme porovnávat pouze dva výsledky. Nejkratší trasu nalezla aplikace amapy.centrum.cz jejíž délka byla 325,7 km. Mapy.cz nalezly trasu dlouhou 328,1 km. Rozdíl mezi nimi činní 2,5 km, což je poměrně malá odchylka, nicméně se nabízí otázka, zda to není mnoho, protože nejkratší cesta by přece měla být jednoznačná.

Pokud se podíváme na výsledky hledání nejrychlejších cest s placenými úseky tak tam byla nalezena nejrychlejší cesta přes mapy.cz, která činí 3 hodiny a 38 minut. Všechny tři nalezené vzdálenosti cest jsou od sebe v rozmezí deseti kilometrů. Aplikace maps.google.cz a amapy.centrum.cz mají velice podobný čas nalezené cesty a to 4 hodiny a 6 minut a 4 hodiny a 2 minuty. Takže můžeme říci, že tyto dvě aplikace našli cestu přibližně se stejnou dobou trvání. Ovšem pokud je porovnáme s nejrychlejší cestou tak je horší zhruba o 25 minut.

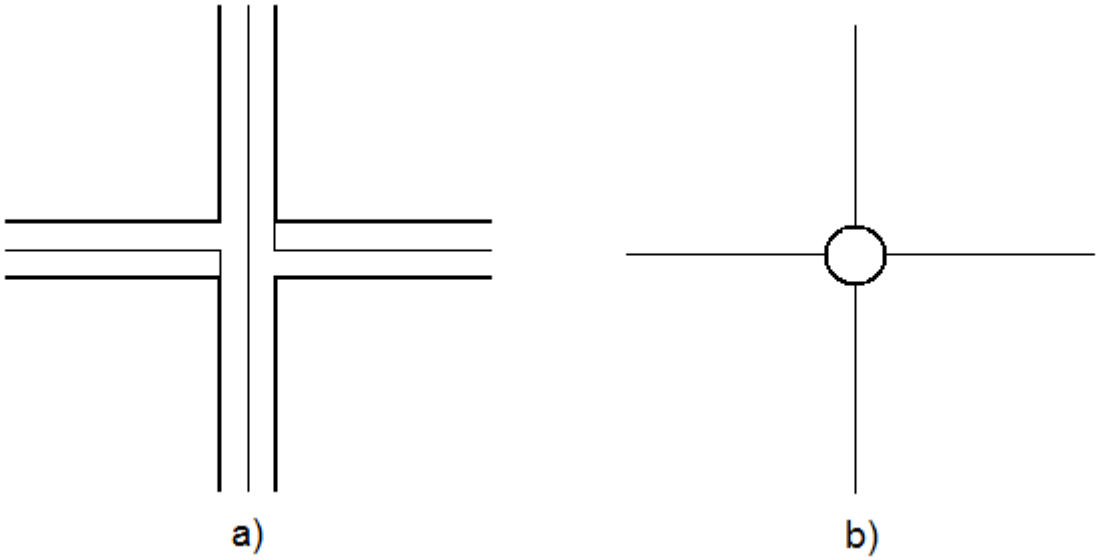
Opačná situace nastává při hledání nejrychlejší cesty bez úseků s poplatky. Sice všechny aplikace najdou cestu, jejich vzdálenosti jsou od sebe v rozmezí šesti kilometrů, ale co se týče časů, tak ty mají velice podobné mapy.cz (4 hodiny a 44 minut) a amapy.centrum.cz (4 hodiny a 33 minut). Nejhorší výpočet času nastává u maps.google.cz tam byl vypočten čas 5 hodin a 57 minut, což v porovnání s nekratším časem dělá rozdíl 84 minut, což je poměrně velký rozdíl.

Na ukázaných případech hledání cest je vidět, že výsledky hledání se více či méně odlišují. Dokonce i při hledání nejkratší cesty, kde se předpokládá, že bude existovat právě jedna. Z toho lze usuzovat, že každá aplikace pro hledání cest používá vlastní způsob hledání. Může se jednat o používání různých algoritmů a jejich nastavení, či to může být způsobeno rozdílností dat, nad kterými se cesta vyhledává.

Aplikace mapy.cz používá mapové data od firmy NAVTEQ, amapy.centrum.cz od společnosti DPA a maps.google.cz od společnosti Tele Atlas.

3 Algoritmy pro směrování

Pokud si představíme, jak vypadá silniční síť, pak je jasné, že jí můžeme reprezentovat orientovaným grafem. V tomto grafu jsou křižovatky uzly grafu a cesty mezi nimi jsou hrany grafu, které jsou orientované podle směru jízdy. Princip převodu reálné silnice na graf je zobrazen na obrázku 3-1, kde obrázek 3-1a ukazuje, jak vypadá reálná křižovatka a na obrázku 3-1b je zobrazení této křižovatky v grafové podobě. Jak je vidět, tak silnice vedou oběma směry. Z tohoto důvodu nejsou zobrazeny žádné šipky ukazující směr v grafu, každá silnice se dá využít v obou směrech.



Obrázek 3-1: a) Zobrazení reálné křižovatky, b) její převedení do grafové podoby

Díky tomuto převodu můžeme pro hledání cest využít algoritmy, které se používají pro hledání cest v grafech. Asi nejznámější algoritmus pro vyhledání nejkratší cesty v grafu je Dijkstrův algoritmus. Tento algoritmus lze použít v grafu (neorientovaném či orientovaném). Pro správnou funkci musí být hrany ohodnoceny jejich délkou, která nesmí být záporná, což v případě kdy graf reprezentuje silniční síť, nenastane.

Dalším algoritmem, který se dá vyžít pro hledání nejkratší cesty je A* (A star) algoritmus. Vychází z Dijkstrova algoritmu a při hledání cesty využívá heuristiku.

Jeho hlavní výhoda je v tom, že při hledání cesty využívá pouze zlomek dat oproti již zmiňovanému Dijkstrovi, tudíž je mnohem rychlejší.

Heuristika spočívá v co nejpřesnějším odhadu vzdálenosti od aktuálního uzlu co cílového uzlu. Na základě toho je cesta přímo vedena od počátečního uzlu k hledanému koncovému uzlu. Neprocházejí se tedy všechny uzly v grafu, ale jen ty na ležící na nejkratší cestě. Přestože nenajde úplně tu nejkratší cestu, za což může právě heuristika, nicméně nalezená cesta se té nejkratší hodně přibližuje.

3.1 Dijkstrův algoritmus

Tento algoritmus vymyslel Edsger Wybe Dijkstra v roce 1956, jak je uvedeno ve [3]. Jednalo se o Nizozemského fyzika a informatika, který se narodil v roce 1930 v Rotterdamu.

Dijkstrův algoritmus slouží k nalezení nejkratší cesty v grafu. Jak již bylo řečeno, tak zásadní podmínkou pro správné fungování algoritmu je skutečnost, že hrany grafu jsou ohodnoceny nezáporným číslem. Pokud bychom chtěli hledat cestu v grafu, kde jsou hrany ohodnoceny i záporně, pak se musí použít Bellman-Fordův algoritmus. Nicméně s takovými grafy se v této práci nesetkáme. Dijkstrův algoritmu je konečný, protože se v každém průchodu cyklu přesune jeden uzel z množiny uzlů, která je nenavštívená, do množiny uzlů, která již byla navštívena. Musíme ovšem také zadat konečný graf, nad kterým se bude hledat nejkratší cesta. Algoritmus končí, až jsou všechny uzly navštíveny. Jeho složitost je v nejhorším případě $O(|V|^2 + |E|)$, kde $|V|$ je počet vrcholů a $|E|$ je počet hran.

3.1.1 Popis algoritmu

Princip hledání nejkratší cesty Dijkstrovým algoritmem je popsán následujícím pseudokódem. Ten se skládá z 16 hlavních kroků algoritmu. Kroky 1-6 se provedou pouze jednou, zatímco kroky 7-16 se provedou minimálně tolikrát kolik je počet vrcholů v zadaném grafu, nad kterým se má cesta vyhledat.

```
1. Funkce Dijkstra(E, V, s):
2.     Pro každý vrchol (v) v grafu (V):
3.         vzdálenost[v] = nekonečno
4.         předchozí[v] = neznámý
5.     vzdálenost[s] = 0
6.     N = V
7.     Dokud N není prázdná:
8.         u = vrchol s nejmenší vzdáleností z N
9.         pokud vzdálenost[u] = nekonečno:
10.            ukonči cyklus
11.        Odstraň u z N
12.        Pro každého souseda v z u
13.            alt = vzdálenost[u] + vzdálenost (u, v)
14.            pokud alt < vzdálenost[v]
15.                vzdálenost[v] = alt
16.                předchozí[v] = u
```

V prvním kroku je definice funkce s názvem Dijkstra, která provádí hledání nejkratší cesty. Tato funkce přijímá několik parametrů označený písmenky *E*, *V* a *s*. Parametr *E* znamená množinu všech hran, které se nacházejí v grafu, nad kterým hledáme již zmiňovanou nejkratší cestu. Další parametr je *V*, což množina všech vrcholů téhož grafu. Jako poslední předávaný parametr je parametr *s*. To je označení pro počáteční vrchol, z něhož se má začít vyhledávat. Nicméně protože hledáme nejkratší cestu, která je mezi dvěma vrcholy, tak potom je zbytečné aby se hledala nejkratší cesta od počátku ke všem vrcholům, ale bude stačit, když se vyhledá nejkratší cesta pouze k určitému cílovému vrcholu. Z tohoto důvodu můžeme přidat další parametr, který se bude do funkce přidávat.

Krok číslo 2 obsahuje cyklus, ve kterém se opakují kroky 3 a 4. Tento cyklu říká, že se budou procházet všechny uzly patřící do množiny *V*, což jsou všechny uzly zadaného grafu. Každému uzlu se nastaví vzdálenost od počátečního vrcholu na nekonečno. To je z důvodu počáteční inicializace proměnných, a navíc tuto vzdálenost ještě neznáme. Bude teprve vypočítána při provádění algoritmu. Taktéž hodnota předchozího uzlu na nejkratší cestě není známá, proto se nastaví na hodnotu neznámá.

V kroku číslo 5 se nastaví hodnota vzdálenosti z počátečního uzlu do počátečního uzlu na hodnotu 0. To se provádí pouze jednou a je to potřeba kvůli správné funkci algoritmu. V dalším kroku se vytvoří množina nenavštívených vrcholů *N* a přiřadí se do ní vrcholy. Z této množiny se budou postupně vybírat vrcholy, které ještě algoritmus neprozkoumal. Jelikož zatím nebyly prozkoumány ještě žádné vrcholy tak se přidají všechny vrcholy z množiny *V*.

Hlavní cyklus algoritmu začíná v kroku 7. Ten se prochází, dokud není množina nenavštívených uzlů *N* prázdná. Jakmile je tato množina prázdná, tak cyklus skončí, což také znamená konec algoritmu. V osmém kroku je vybrán vrchol z množiny nenavštívených vrcholů, který má nejkratší vzdálenost do počátečního vrcholu. S tímto vrcholem se bude dále pracovat.

Krok číslo 9 obsahuje ošetření případu, kdy je vybrán jako vrchol s nejkratší vzdáleností takový vrchol, jehož vzdálenost do počátečního uzlu je nekonečno. V takovém případě už neexistuje žádný další vrchol, do kterého se dostaneme z počátečního vrcholu, tudíž se algoritmus ukončí. To zaručuje krok 10. Takový vrchol, do kterého nevede cesta z počátečního uzlu, si můžeme představit, jako vrchol ze kterého vedou hrany směrem ven, ale žádná hrana směrem dovnitř.

V této části algoritmu taky můžeme ověřovat, zda nalezený uzel není koncový, pokud jsme ho předali jako parametr funkci. Jestliže je koncový, pak se už dále prohledávat nemusí a algoritmus se může ukončit, protože nejkratší cesta byla právě nalezena.

Takže aktuální vrchol s nejkratší vzdáleností, který jsme vybrali v kroku 8, je nalezen a bude se s ním dále pracovat. Nejdříve ale musíme tento vrchol odstranit z množiny nenavštívených vrcholů. To je provedeno v kroku 11.

V kroku číslo 12 je cyklus na procházení všech sousedů aktuálně vybraného uzlu s nejkratší vzdáleností. Pro každého nalezeného souseda se postupně provedou kroky 13-16.

Krok 13 vypočítá nově nalezenou vzdálenost do nalezeného souseda aktuálního uzlu. Ta se vypočítá tak, že provede součet vzdálenosti aktuálního uzlu s délkou hrany od aktuálního uzlu k jeho nalezenému sousedovi. Vypočtená hodnota se v dalším kroku porovná s aktuální hodnotou vzdálenosti uloženou v sousedovi. Jestliže je nalezená hodnota menší než aktuální pak se provedou následující kroky. Vzdálenost se přepíše na nově nalezenou hodnotu a taktéž se změní soused, přes kterého jsme toho hodnotu nalezli. To je provedeno v krocích 15 a 16. Pokud uzel má aktuální vzdálenost s hodnotou nekonečno, což znamená, že ještě žádná cesta nebyla nalezena do tohoto uzlu, tak se vždycky přepíše právě nalezenou vzdáleností.

3.1.2 Příklad na vyhledání nejkratší cesty

V předchozí kapitole jsme si ukázali princip Dijkstrova algoritmu. Z pouhého popisu však nemusí být na první pohled zcela jasné, jak algoritmus skutečně pracuje. Proto je v této kapitole ukázán postup Dijkstrova algoritmu v grafu.

Na obrázku 3-2 je zobrazen již zmiňovaný postup Dijkstrova algoritmu. V tomto obrázku je postupně zobrazeno šest kroků, které se provádí při hledání nejkratší cesty nad uvedeným grafem. Tento graf obsahuje pět vrcholů a šest neorientovaných hran. V každém vrcholu je uvedeno číslo, které představuje nejkratší vzdálenost do počátečního vrcholu, čili do toho vrcholu ve kterém se začalo vyhledávat. Každá hrana má taktéž přiřazeno číslo značící délku hrany.

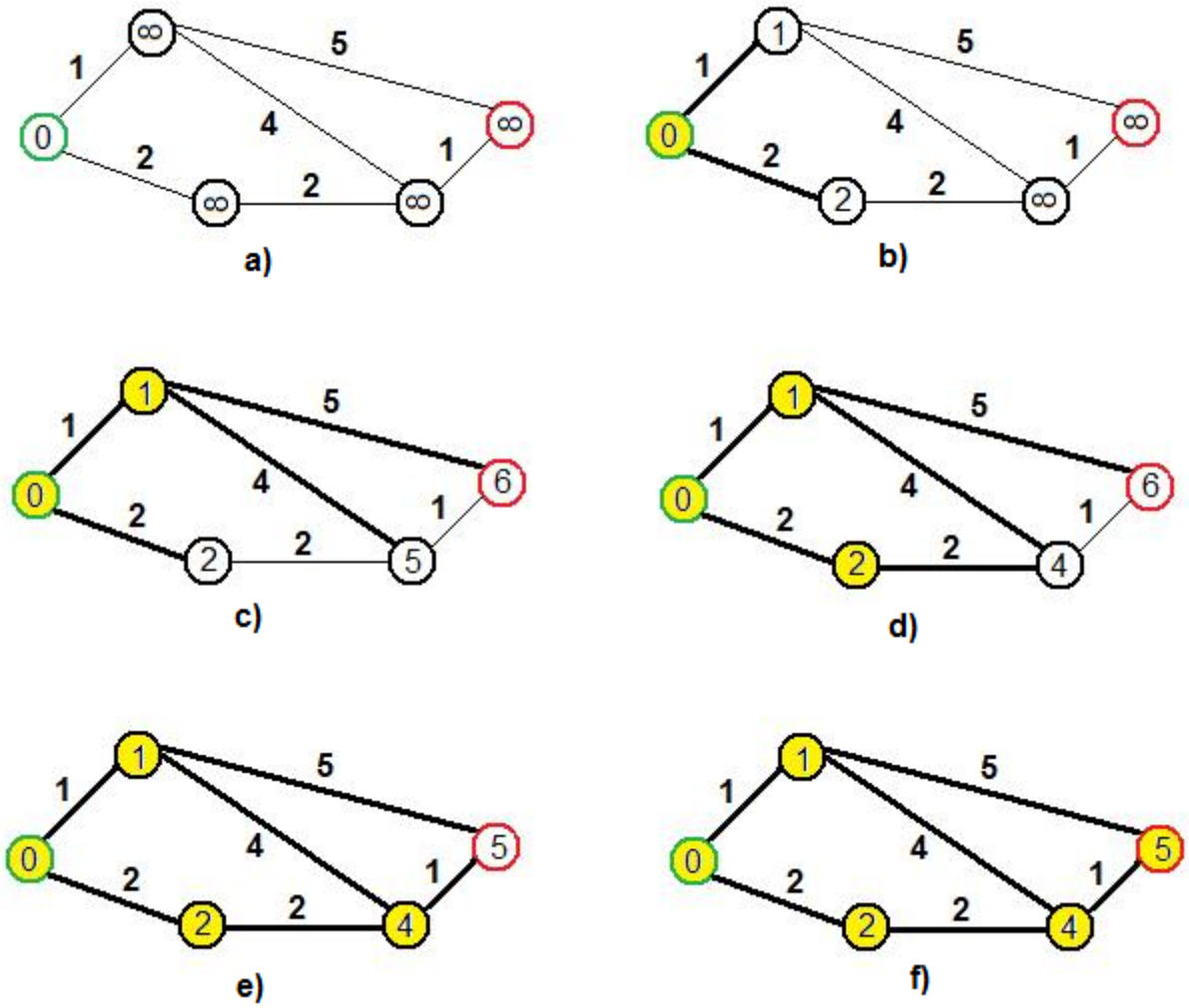
Při provádění algoritmu se mnění hodnoty v jednotlivých vrcholech a to jen v případě, že nalezená vzdálenost je kratší, než aktuální vzdálenost uložená ve vrcholu. Dále pro větší přehlednost v již zmiňovaném obrázku je prozkoumaná hrana zobrazena tučně. Žlutou barvou se značí vrcholy, které již algoritmus prošel, a které se už znova projít nesmí. Počáteční vrchol je zobrazen zelenou barvou a koncový vrchol barvou červenou.

Začátek algoritmu je zobrazen na obrázku 3-2a. Jak je vidět, tak již proběhla úvodní fáze inicializace vrcholů, kdy se ke všem vrcholům přiřadí vzdálenost nekonečno a k počátečnímu vrcholu se přidá vzdálenost nula. Následný krok je zobrazen na obrázku 3-2b. Vybere se vrchol s nejkratší vzdáleností od počátečního vrcholu. V tomto případě to je počáteční uzel s hodnotou vzdálenosti 0. U tohoto vrcholu se prozkoumají jeho sousedé a vypočte se jim vzdálenost. Poté dojde k porovnání vzdáleností a k uložení té lepší. Počáteční vrchol má dva sousedy a to přes hrany s délkou 1 a 2. Vzhledem k tomu, že v sousedech jsou uloženy hodnoty nekonečno, což oznamuje, že cesta do těchto vrcholů nebyla ještě nalezena, tak se přepíší hodnoty vzdáleností v těchto uzlech. Hodnoty vzdáleností v těchto uzlech jsou dány pouze vzdáleností hrany od počátečního uzlu. Počáteční uzel se označí jakou použitý, je označen žlutou barvou. Algoritmus dokončil první iteraci a může přejit k další.

Další krok je zobrazen na obrázku 3-2c. Je opět potřeba vybrat vrchol s nejkratší vzdáleností, což je vrchol se vzdáleností 1. Tento vrchol má dva sousedy přes hrany se vzdáleností 4 a 5. oba sousedi mají vzdálenost opět nekonečnou, tudíž se přepíše vzdálenost na aktuální nalezené hodnoty vzdálenosti a to na 5 a 6. V tomto kroku jsme sice našli vzdálenost do koncového uzlu, ale vzhledem k tomu, že jsou ještě neprozkoumané uzly, tak nemůže algoritmus skončit. Není zaručeno, že nalezená cesta je ta nejkratší.

Na obrázku 3-2d je následující krok algoritmu. Dále je vybrán vrchol se vzdáleností 2. Tento vrchol má pouze jednoho ještě nepoužitého souseda. Soused má hodnotu nejkratší vzdálenosti 5, nicméně aktuální nalezená vzdálenost má hodnotu 4, proto bude stará hodnota vzdálenosti nahrazena novou.

Pátý krok je na obrázku 3-2e a je vněm provedena změna vzdálenosti u koncového vrcholu, která se změnila z hodnoty 6 na 5. Graf po skončení algoritmu je na obrázku 3-2f. Všechny uzly jsou použity, tudíž algoritmus končí a hodnota v koncovém vrcholu je nejkratší nalezená vzdálenost, jakou se dá do vrcholu dostat. V tomto případě je nejkratší vzdálenost 5.



Obrázek 3-2: Postup Dijkstrova algoritmu

3.2 A* algoritmus

A* algoritmus slouží pro nalezení nejkratší cesty v grafu. Vychází z Dijkstrova algoritmu a doplňuje ho o heuristiku, která umožňuje vyhledávání v mnohem kratším čase a s menším objemem dat ve kterých se vyhledává. Tento algoritmus byl poprvé popsán v roce 1968 několika americkými vědci, jak je uvedeno v [1].

Heuristika spočívá v tom, že algoritmus pro každý nalezený vrchol odhaduje vzdálenost od aktuálně nalezeného vrcholu do koncového vrcholu. Tuto vzdálenost přičítá k vypočtené vzdálenosti z počátečního vrcholu do aktuálního vrcholu. A podle takto vypočtené hodnoty se dále rozhoduje, do kterého vrcholu půjde v dalším kroku. Vybírá vždy nejkratší takto nalezenou vzdálenost. Díky tomu algoritmus při vyhledávání cesty směřuje přímo do cílového vrcholu. Nemůže tedy nastat situace, že by se procházely vrcholy, které leží na opačné straně, než se nachází cílový vrchol.

Pro správnou funkci algoritmu je nezbytně důležitá funkce, která odhadne vzdálenost mezi dvěma vrcholy grafu. Mluvíme o odhadu, protože u velkých grafů není možné mít uloženo v každém vrcholu informace o vzdálenosti do každého dalšího vrcholu. Hlavním důvodem jsou velké paměťové

nároky. Pokud si představíme graf s milióny uzly, pak by v každém vrcholu muselo být uloženo milion údajů o vzdálenosti. Pokud bychom počítali, kolik by bylo celkově uloženo takových hodnot pro celý graf, tak se dostáváme k obrovskému číslu. Proto je mnohem výhodnější vytvořit funkci, která bude onu vzdálenost odhadovat i za podmínky, že tato odhadnutá vzdálenost nebude zcela přesná. Paměť, kterou tím ušetříme, bude značná, hlavně pokud se bude jednat velký graf.

Na obrázku 3-3 je zobrazeno porovnání použitých dat při hledání nejkratší cesty Dijkstrovým algoritmem (červeně)a A* algoritmem (zeleně) mezi Prahou a Ostravou. Dijkstrův algoritmus hledá vždy nejkratší cestu do všech vrcholů, tudíž prochází všechny vrcholy postupně podle vzdálenosti od nejkratší po nejdelší. Prochází tedy i vrcholy, která leží na opačné straně než je cílový vrchol. Naproti tomu A* algoritmus prochází pouze vrcholy, které směrují od počátku k cíli. Z tohoto hlediska je množství procházených dat mnohem menší než u Dijkstrova algoritmu a tudíž algoritmus rychlejší.

Obrázek 3-3: Použitá data při vyhledávání nejkratší cesty mezi Prahou a Ostravou pomocí A* (zeleně) a Dijktrovým algoritmem (červeně)

3.2.1 Popis algoritmu

V této kapitole je zobrazen pseudokód popisující princip činnosti A* algoritmu. Kroky 1 až 6 se provedou pouze jednou, zatímco kroky 7 až 28 jsou hlavní kroky algoritmu a provedou se minimálně tolikrát, kolika vrcholy se bude procházet při hledání cesty.

1. Funkce A* (***začátek***, ***konec***)
2. množina prozkoumaných vrcholů = prázdná
3. množina neprozkoumaných vrcholů = počáteční uzel
4. **v_skóre**[***začátek***] = 0
5. **o_skóre**[***začátek***] = odhadnutá vzdálenost (***začátek, konec***)
6. **c_skóre**[***začátek***] = **v_skóre**[***začátek***] + **o_skóre**[***začátek***]
7. Dokud není množina neprozkoumaných vrcholů prázdná
8. ***x*** = vrchol s nejnižším **c_skóre[]** z neprozkoumaných vrcholů
9. Pokud ***x*** = ***konec***
10. cesta byla nalezena, konec algoritmu
11. odstranění uzlu ***x*** z neprozkoumaných vrcholů

12. přidání uzlu ***x*** do prozkoumaných vrcholů
13. Pro každého souseda ***y*** vrcholu ***x***
14. Pokud je ***y*** v množině prozkoumaných vrcholů
15. přejdi k dalšímu sousedovi ***x***, krok 13
16. **aktuální vzdálenost = v_skóre[*x*] + vzdálenost (*x*, *y*)**
17. Pokud ***y*** není v neprozkoumané množině vrcholů
18. tak ho přidej do neprozkoumané množiny vrcholů
19. aktuální vzdálenost je nejlepší = true
20. Jinak když **aktuální vzdálenost < v_skóre[*y*]**
21. aktuální vzdálenost je nejlepší = true
22. Jinak
23. aktuální vzdálenost je nejlepší = false
24. Pokud aktuální vzdálenost je nejlepší == true
25. **přišel_z[*y*] = *x***
26. **v_skóre[*y*] = aktuální vzdálenost**
27. **o_skóre[*y*]** = odhadnutá vzdálenost (***y***, ***konec***)
28. **c_skóre[*y*] = v_skóre[*y*] + o_skóre[*y*]**

Název funkce a předávané parametry jsou uvedeny v prvním kroku algoritmu. Mezi předávanými parametry je počáteční vrchol, kde se má začít cesta vyhledávat a koncový vrchol, do kterého se má cesta nají. Dalším předávaným parametrem může být graf, nad kterým bude algoritmus pracovat, nicméně v tomto případě se nepředává.

V kroku číslo 2 je vytvořena množina, do které se budou ukládat vrcholy, které již algoritmus prošel. V tuto chvíli je tato množina zcela prázdná, algoritmus totiž ještě nezačal pracovat. Krok číslo 3 vytváří množinu neprozkoumaných vrcholů, což jsou vrcholy, které byly již nalezeny, ale ještě nebyly prozkoumány.

V krocích číslo 4-6 se provádí výpočet tří různých typů vzdáleností. První z nich je vzdálenost v_skóre[začátek]. Tato vzdálenost je aktuální vzdálenost od počátečního vrcholu. V tomto případě je to 0, jelikož se jedná o počáteční vrchol. Další vzdáleností je o_skóre[začátek], což je odhadnutá vzdálenost z aktuálního vrcholu do cílového vrcholu. Způsob realizace funkce pro odhad vzdálenosti je uveden v kapitole 3.2.2. Posledním typem vzdálenosti, která je potřeba, je c_skóre[začátek]. Tato vzdálenost představu celkovou vzdálenost mezi počátečním a koncovým vrcholem a vypočítá se součtem v_skóre[začátek] a o_skóre[začátek], neboli součtem aktuální vzdáleností od počátečního vrcholu a odhadnutou vzdáleností do koncového vrcholu.

Krok číslo 7 již obsahuje hlavní cyklus algoritmu, který se provádí, dokud není množina neprozkoumaných vrcholů prázdná. V osmém kroku se provádí výběr vrcholu, který má nejmenší hodnotu c_skóre[]. Můžeme také říct, že se vybírá vrchol, přes který v daném okamžiku vede cesta s nejkratší vzdáleností mezi počátečním a koncovým vrcholem. Tento vybraný vrchol se bude v následujících krocích algoritmu dále zpracovávat.

Kroky číslo 9 a 10 říkají, kdy může algoritmus předčasně skončit ještě před dokončením hlavního cyklu. Takové skončení nastává v případě, když je jako vrchol, který se bude procházet, vybrán cílový vrchol. V takovém případě jsme našli nejkratší cestu a není dále potřeba procházet graf. Stejně už kratší cestu nenalezneme, i kdybychom dále nechali hlavní cyklus algoritmu běžet.

V krocích 11 a 12 proběhne přesunutí aktuálně vybraného vrcholu z množiny neprozkoumaných vrcholů do množiny prozkoumaných vrcholů.

V kroku číslo 13 je další cyklus, který prochází všechny sousedy aktuálního uzlu a provádí pro ně kroky 14 až 28. V krocích 14 a 15 se provádí ošetření případu, kdy je nalezený soused již v množině prozkoumaných vrcholů. V takovém případě již byl soused prozkoumán a algoritmus pokračuje výběrem dalšího souseda aktuálního uzlu, což je krok číslo 13.

V kroku číslo 16 se provede výpočet aktuální vzdálenosti nalezeného souseda od počátečního vrcholu. Ten se získá součtem vzdálenosti aktuálního vrcholu a vzdáleností mezi jeho sousedem. Dále následuje blok podmínek, kroky 17-23, kde se porovnává právě nalezená aktuální vzdálenost.

Krok číslo 17 ověřuje, jestli se aktuálně nalezený soused nachází v množině neprozkoumaných vrcholů. Pokud se tam nenachází, tak se tam v kroku 18 přidá. Dále se v kroku 19 provede nastavení proměnné, aktuální vzdálenost je nejlepší, na hodnotu pravda. To je z toho důvodu, že doposud do tohoto souseda nebyla nalezena žádná cesta a tudíž nalezená vzdálenost bude ta nejkratší.

Pokud podmínka v kroku číslo 17 není splněna, tak se provede podmínka kroku číslo 20. Ta ověří, zda aktuální nalezená vzdálenost je menší než vzdálenost uložená v aktuálním sousedovi vrcholu. Pokud je tato podmínka splněna, tak se v kroku číslo 21 provede nastavení proměnné, aktuální vzdálenost je nejlepší, na hodnotu pravda.

Jestliže podmínky v krocích číslo 17 a 20 nejsou splněny, tak se provedou kroky číslo 22 a 23, které provedou nastavení proměnné, aktuální vzdálenost je nejlepší, na hodnotu nepravda.

Krok číslo 24 obsahuje podmínku porovnávající, jestli je pravdivá hodnota uložená v proměnné, aktuální vzdálenost je nejlepší. Pokud je, tak se provedou následující kroky číslo 25-28. V kroku číslo 25 se uloží v sousedovi informace, z kterého vrcholu jsme k němu dostali. Krok číslo 26 obsahuje uložení aktuální nalezené vzdálenosti do právě procházeného souseda vrcholu.

Krok číslo 27 dále vypočítá odhadnutou vzdálenost z právě procházeného souseda do koncového vrcholu a uloží ji. Poslední krok číslo 28 vypočte celkovou vzdálenost cesty, pokud půjdeme přes právě procházeného souseda. Jakmile skončí krok číslo 28, tak se pokračuje dále v kroku číslo 13, případně číslo 7.

Algoritmus končí v případě, když se dostal ke koncovému vrcholu nebo pokud je množina neprozkoumaných vrcholů prázdná. Pokud se dostal ke koncovému vrcholu, pak nalezl hledanou cestu, nicméně pokud prošel všechny neprozkoumané vrcholy bez toho, aby se dostal do cílového vrcholu, pak hledaná cesta nebyla nalezena.

3.2.2 Způsob výpočtu odhadnuté vzdálenosti

V předchozím popisu principu fungování algoritmu A* jsme se nijak nezabývali výpočtem odhadnuté vzdálenosti mezi dvěma vrcholy. Prostě jsme ji brali jako hodnotu, kterou není problém zjistit a počítat s ní. Nicméně právě přesnost odhadnutí vzdálenosti mezi jakýmikoli dvěma vrcholy je klíčová věc, na které je tento algoritmus postaven.

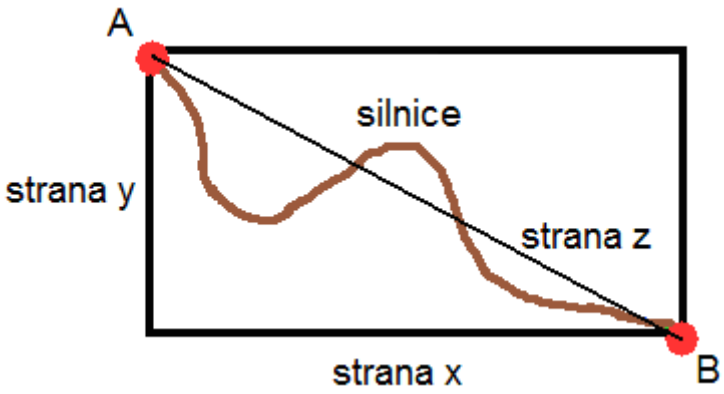
Je zapotřebí poznamenat, že není možné znát opravdovou skutečnou vzdálenost mezi každými dvěma vrcholy. To by znamenalo uchovávat informace u každého vrcholu o vzdálenosti ke všem ostatním vrcholům.

Pro příklad si vezměme graf s milionem vrcholů. V takovém případě by se musely ke každému vrcholu uložit informace o vzdálenosti k milionu sousedů. Což by bylo velice paměťově náročné a taky by trval samotný výpočet těchto vzdáleností.

Proto je mnohem lepší řešení tuto vzdálenost odhadnout. V tomto právě spočívá heuristika celého algoritmu. Čím více se odhadnutá vzdálenost přibližuje skutečné vzdálenosti, tím dává algoritmus A* přesnější nalezenou nejkratší cestu.

Výpočet vzdálenosti se provádí pomocí GPS souřadnic obou bodů. Ze získaných GPS souřadnic můžeme získat údaje potřebné pro výpočet vzdálenosti (strany x a y) a dále pomocí Pythagorovi věty vypočítat vzdálenost mezi dvěma body, jak je uvedeno na obrázku číslo 3-4.

Takto vypočítaná vzdálenost, je přímá (letecká), ovšem neodpovídá skutečné vzdálenosti mezi uzly. Skutečná vzdálenost mezi uzly bude o něco větší než vypočtená vzdálenost. Je tedy potřeba tuto vypočtenou leteckou vzdálenost upravit tak aby se co nejvíce podobala skutečné vzdálenosti.



Obrázek 3-4: Výpočet vzdálenosti pomocí Pythagorovi věty

Největší problém se skrývá právě ve výpočtu opravdové vzdálenosti mezi body tj. vzdálenost po silnici od jednoho bodu k druhému. Díky tomu, že známe leteckou vzdálenost mezi těmito dvěma body, tak víme nejkratší možnou vzdálenost mezi nimi, které ovšem nebude skutečnou.

Takže můžeme leteckou vzdálenost vynásobit nějakou konstantou, která zajistí, že délka mezi těmito body se bude více blížit skutečnosti. Díky vynásobení je zaručena přímá úměra mezi skutečnou a leteckou vzdáleností, která by u pouhého přičtení nebyla.

Jako příklad uveďme dvě vypočtené vzdálenosti mezi nějakými dvěma vrcholy, které mají hodnotu 100 km a 20 km. Tato vzdálenost je vypočtena jako přímá neboli letecká. Nyní pokud chceme převést nalezenou leteckou vzdálenost na odhad skutečné vzdálenosti, tak potřebujeme znát konstantu. Dejme tomu, že je tato konstanta 1,3. Potom po vynásobení leteckých vzdáleností touto konstantou dostaneme odhadnutou skutečnou vzdálenost, která má hodnotu 130 km a 26 km. Tyto hodnoty odhadnuté vzdálenosti se přibližují skutečné vzdálenosti více než pouhé vypočtení letecké vzdálenosti.

Pro správné nalezení konstanty je ovšem potřeba experimentovat s různou velikostí konstanty. Tyto experimenty jsou uvedeny v kapitole 6-2.

3.2.3 Příklad na vyhledání nejkratší cesty

Na obrázku 3-5 je zobrazen postup hledání nejkratší cesty v grafu pomocí A* algoritmu. Tento obrázek je rozdělen na pět dalších obrázků, ve kterých jsou zobrazeny jednotlivé kroky algoritmu.

Počáteční vrchol je zobrazen zelenou barvou a koncový vrchol barvou červenou. Prozkoumané vrcholy jsou vybarveny žlutou barvou. U každé hrany je zobrazena její délka a tučnou čarou jsou vyznačeny ty hrany, které byly prozkoumány. U nalezených vrcholů je uvedena jejich odhadnutá vzdálenost do koncového vrcholu. Tato vzdálenost je označena H(v) a je vypočítána na základě přímé neboli letecké vzdálenosti od aktuálního vrcholu do koncového vrcholu.

Graf, nad kterým se bude vyhledávat, je zobrazen na obrázku 3-5a. Na začátku algoritmu se pouze nastaví aktuální vzdálenost na 0 a provede se odhad vzdálenosti do koncového vrcholu, která má hodnotu 5.

Další krok je zobrazen na obrázku 3-5b. Provede se výběr vrcholu s nejkratší celkovou vzdáleností mezi počátečním a koncovým vrcholem. Protože je zatím nalezen jen počáteční uzel tak bude právě on vybrán. Prozkoumají se sousední vrcholy počátečního vrcholu a uloží se hodnoty jejich vzdáleností od počátečního vrcholu. V tomto případě jsou nalezeni dva sousedi, jejichž vzdálenost je 2 a 3. Přidají se do množiny neprozkoumaných vrcholů a vypočtou se jejich odhadované vzdálenosti do koncového vrcholu. Aktuální vrchol se odebere z množiny neprozkoumaných vrcholů a přidá se do množiny prozkoumaných vrcholů. Tímto je tento krok ukončen a přechází se k dalšímu výběru vrcholu.

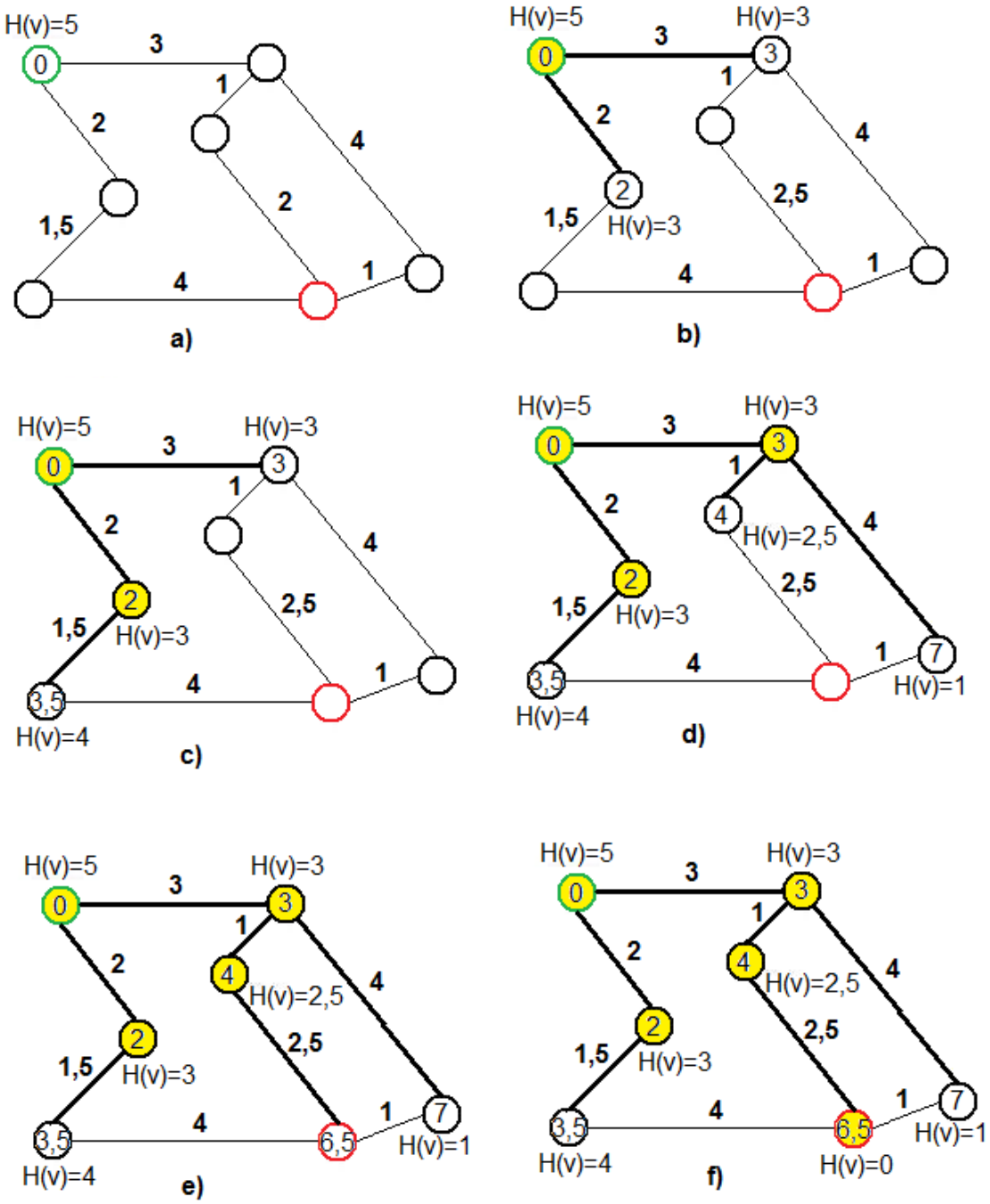
Druhý krok algoritmu je zobrazen na obrázku 3-5c. Opět musíme provést výběr nejkratší celkové vzdálenosti mezi počátečním a koncovým vrcholem. Tato vzdálenost se počítá jako součet aktuální vzdálenosti od počátečního vrcholu a odhadnuté vzdálenosti z aktuálního vrcholu do koncového vrcholu.

V množině neprozkoumaných vrcholů se nacházejí dva vrcholy s aktuálními délkami od počátku 2 a 3. Celková vzdálenost 5 je pro vrchol s aktuální vzdáleností 2 a pro druhý vrchol s aktuální vzdáleností 3 je celková vzdálenost 6. Proto je jako vrchol s nejkratší celkovou vzdáleností vybrán vrchol s aktuální vzdáleností 2. Nyní se prozkoumají jeho sousedi, což je v tomto případě pouze jeden soused. Vypočte se aktuální vzdálenost, jež má hodnotu 3,5 a dále se provede odhad vzdálenosti ke koncovému vrcholu, který má hodnotu 4.
Aktuální vrchol se přesune z množiny neprozkoumaných vrcholů do množiny prozkoumaných vrcholů a přejde se k dalšímu kroku.

Ten je zobrazen na obrázku 3-5d. V množině neprozkoumaných vrcholů se nachází opět dva vrcholy. První má aktuální vzdálenost 3 a celkovou vzdálenost 6 a ten druhý má aktuální vzdálenost 3,5 a celkovou vzdálenost 7,5. Nejnižší celková vzdálenost je 6, a proto bude jako aktuální vrchol vybrán vrchol s aktuální vzdáleností 3. Tento vrchol má dva sousedy se vzdáleností 1 a 4.

Provede se výpočet aktuálních vzdáleností, které mají hodnotu 4 a 7. Dále se provede výpočet odhadu vzdáleností nalezených sousedů do koncového vrcholu a sousedi se přidají do množiny neprozkoumaných vrcholů. Aktuální vrchol se přesune z množiny neprozkoumaných vrcholů do množiny prozkoumaných vrcholů a přejde se k dalšímu kroku.

Další krok je na obrázku 3-5e. Nyní se nachází v množině neprozkoumaných vrcholů tři vrcholy. Tyto vrcholy mají celkové vzdálenosti hodnoty 7,5, 6,5 a 8. Jako aktuální vrchol se vybere vrchol s aktuální vzdáleností 4, který má nejmenší celkovou vzdálenost, jenž má hodnotu 6,5.



Obrázek 3-5: Postup A* algoritmu

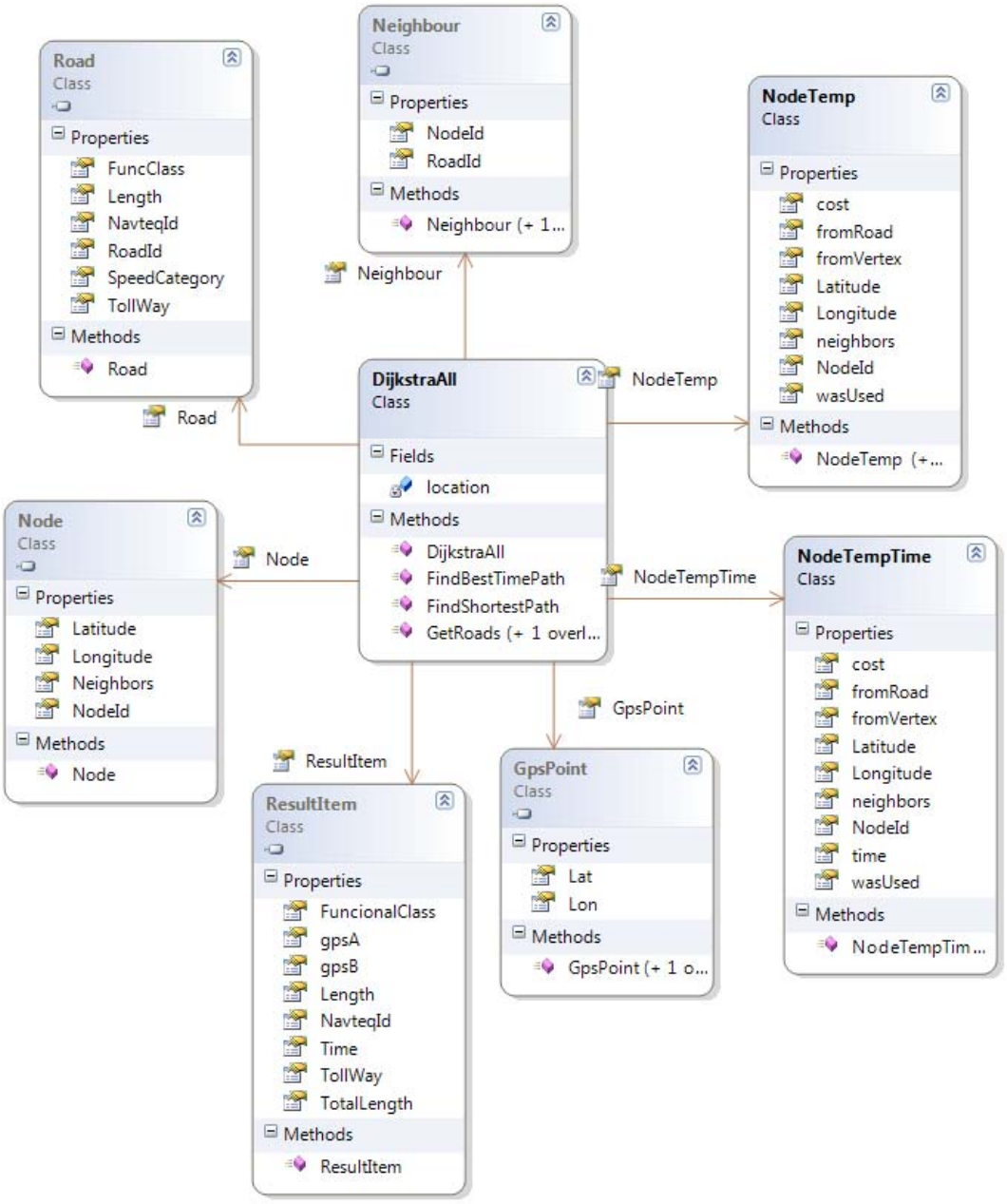
K tomuto vrcholu se prozkoumá jeho soused, který je vzdálen 2,5. Vypočítá se aktuální vzdálenost od počátečního vrcholu, která má hodnotu 6,5 a také se vypočítá odhadovaná vzdálenost do koncového vrcholu. Provede se přesunutí aktuálního vrcholu z neprozkoumané množiny do prozkoumané a přejde se k poslednímu kroku.

Poslední krok je zobrazen na obrázku 3-5f. V množině neprozkoumaných vrcholů se nachází vrcholy s celkovými vzdálenostmi 7,5, 8 a 6,5. Je vybrán vrchol s nejkratší vzdáleností 6,5. Tento vrchol je koncový, a tudíž algoritmus končí. Nejkratší cesta byla nalezena se vzdáleností 6,5.

4 Implementace Dijkstrova algoritmu

V této kapitole se budeme zabývat implementací Dijkstrova algoritmu a také pomocnými třídami, které jsou potřeba pro jeho správnou funkci.

Diagram tříd Dijkstrova algoritmu je zobrazen na obrázku 4-1. Třída *DijsktraAll* se nachází uprostřed diagramu a obsahuje funkce pro výpočet nejkratší a nejrychlejší cesty. Ostatní třídy kolem jsou pomocné třídy, jejichž funkce jsou vysvětleny v následující části textu.



Obrázek 4-1: Diagram tříd Dijkstrova algoritmu

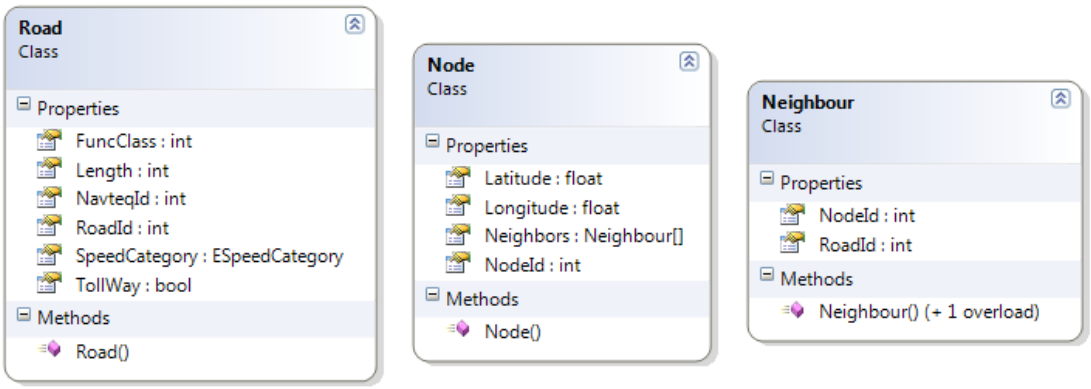
4.1 Pomocné třídy Dijkstrova algoritmu

Tyto třídy představují různé entity, které jsou potřeba pro algoritmus. Nejčastěji tyto entity představují nějaké reálné objekty jako například uzel grafu, hranu grafu či GPS souřadnici. Dále pak slouží k nadefinování, v jakém formátu se budou vracet výsledky o nalezených cestách.

4.1.1 Třídy reprezentující data grafu – Node, Road a Neighbour

Na obrázku 4-2 jsou tyto třídy zobrazeny včetně proměnných a jejich datových typů. Každá z těchto tříd taktéž obsahuje metodu, která slouží k naplnění dat.

Jak už bylo řečeno, tak tyto třídy představují graf, nad kterým se bude vyhledávat cesta. Při vyhledávání cesty si algoritmus vytváří objekty z těchto tříd, do kterých ukládá načtená data.



Obrázek 4-2: Třídy Road, Node a Neighbour

Třída *Road* představuje hranu grafu, což je vlastně v silniční síti cesta spojující dvě křižovatky. K cestě jsou přiřazeny následující informace. *RoadId* a *NavteqID* jsou identifikační čísla cesty, která jsou jedinečná. Samozřejmostí je uvedení délky cesty (*Length*)
a informace o tom zda je daný úsek placený (*TollWay*). Následují informace o typu cesty (*FuncClass*) a rychlostním limitu cesty (*SpeedCategory*). Tyto informace jsou potřeba hlavně při hledání nejrychlejší cesty, kde se díky nim vypočítá čas projetí cesty. Při hledání nejkratší cesty se bere v potaz pouze délka.

Další třída reprezentuje uzel grafu neboli křižovatku v silniční síti a má název *Node*. Tato třída taktéž obsahuje jedinečné identifikační číslo uzlu (*NodeId*). Ke každému uzlu je uložena jeho gps pozice (Latitude, Longitude).

Dále pak každý uzel obsahuje pole objektů typu *Neighbour*, což jsou informace o jeho sousedech. K uzlu je přiřazeno několik objektů třídy *Neighbour* a to podle toho kolik má sousedních uzlů. O každém sousedovi se dozví jeho identifikátor a také identifikátor cesty, přes kterou se k němu dostane.

4.1.2 Třídy NodeTemp a NodeTempTime

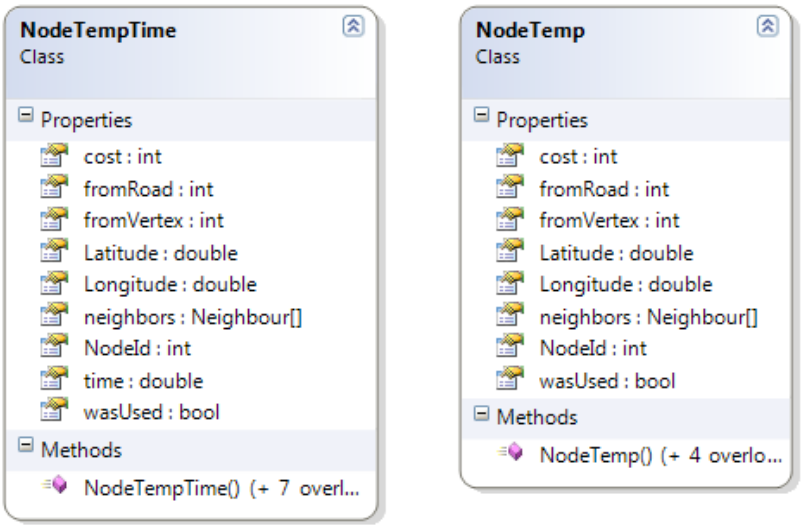
Hlavním účelem těchto tříd je poskytnout jednotnou strukturu, do které se načítají data o grafu a také slouží k ukládání průběžných výpočtů algoritmu. Jejich proměnné a metody jsou zobrazeny na obrázku 4-3.

Obě třídy obsahují stejné proměnné, nicméně třída *NodeTempTime* má o jednu proměnnou více. Ta se jmenuje *time* a slouží k ukládání časové délky cesty. Vzhledem k tomu, že se při hledání cesty Dijkstrovým algoritmem prochází velké množství uzlů, tak je snaha o to aby se minimalizovala potřebná velikost paměti pro uchování jednoho uzlu. Z tohoto důvodu se při hledání nejrychlejší cesty používá pro ukládání dat o uzlech třída *NodeTempTime*, která obsahuje již zmiňovanou proměnnou time. Pro hledání nejkratší cesty se používá třída *NodeTemp*, která je o tuto proměnnou ochuzena. Nicméně v tomto případě se taky zjišťuje celkový čas jízdy, ale ten je vypočten při rekonstruování nalezené cesty.

Nyní se dostáváme k popisu jednotlivých proměnných třídy. Jak už bylo řečeno tak se jedná o proměnné, které jsou informace o uzlu, mezi ně patří *NodeId*, *Latitude*, *Longitude* a *Neighbors*, které jsou popsány v kapitole 4.1.1 a dále se vysvětlovat nebudou.

Mezi další patří takové proměnné, které jsou potřeba pro samotný algoritmus. Atribut *cost* se používá k ukládání aktuální nalezené vzdálenosti v metrech od počátečního vrcholu. Hraje velice důležitou roli při hledání nejkratší cesty. Proměnná *time* slouží k ukládání času potřebného k dosažení uzlu z počátečního uzlu.

Proměnné *fromRoad* a *fromVertex* slouží k uchování informace z jakého uzlu a přes jakou cestu jsme se dostali do aktuálního uzlu. Ty jsou potřebné k následné rekonstrukci nalezené cesty. Posledním velice důležitým atributem je *wasUsed*, které slouží k oznámení, zda již algoritmus tento uzel prozkoumal.



Obrázek 4-3: Třídy NodeTemp a NodeTempTime

4.1.3 Třídy ResultItem a GpsPoint

U každého vrcholu jsou uvedeny jeho GPS souřadnice a to hlavně z důvodu pozdějšího vykreslení nalezené cesty do mapy. GPS souřadnice se skládá ze dvou hodnot určující polohu a to délky a šířky. Předávat polohu uzlu musíme tedy pomocí dvou hodnot, což může být někdy nepřehledné.

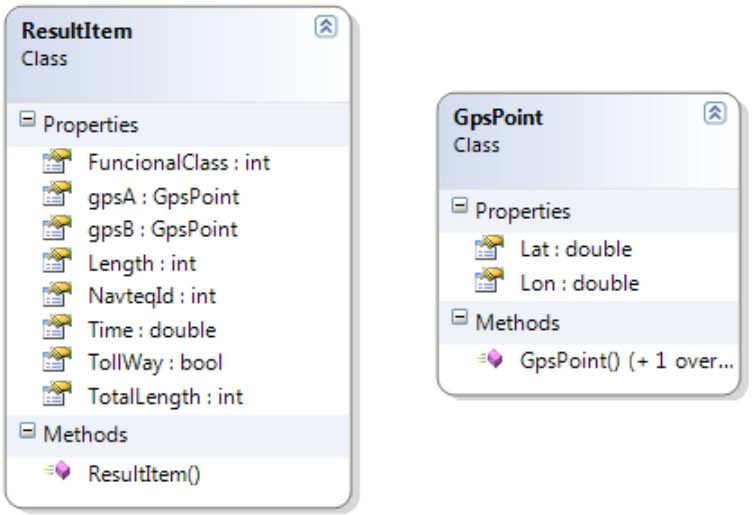
Z tohoto důvodu je vytvořena třída *GpsPoint*, která reprezentuje gps souřadnici. Tato třída je ukázána na obrázku 4-4. Jejími proměnnými jsou Lat a Lon, jež představují souřadnice GPS. Pokud tedy je potřeba předat GPS souřadnice, pak se vytvoří právě z této třídy objekt, do něhož se uloží souřadnice a který se pak dále pracuje.

Pro výstup algoritmu hledání cesty se používá třída *ResultItem*. Ta obsahuje informace o nalezených silnicích, přes které vede požadovaná nejkratší či nejrychlejší cesta. Takže jako výstup z algoritmu se předává kolekce objektů této třídy.

Proměnná *FuncionalClass* o jaký typ cesty se jedná. Číslem 1 jsou označeny dálnice a nejhorší a nejpomalejší cesty jsou označeny číslem 5. *TollWay* říká, zda je úsek cesty placený či neplacený.

Pro vykreslování do mapy slouží proměnné *gpsA* a *gpsB*. Obě jsou typu GpsPoint a jsou v nich uloženy gps souřadnice. *GpsA* je poloha začátku cesty a *GpsB* je poloha konce cesty. *Length* a *TotalLength* jsou proměnné, ve kterých se nacházejí vzdálenosti cest v metrech. Délka aktuálního úseku cesty je uložena v *Length* a celková délka nalezené cesty je uložena v proměnné *TotalLength*.

Pro uložení celkového času nalezené cesty je použita proměnná *Time*. Tento čas je uložen v sekundách. Posledním proměnnou je *NavteqId* což je pouze jednoznačný identifikátor a prakticky nemá žádný další význam.



Obrázek 4-4: Třídy ResultItem a GpsPoint

4.2 Princip implementace důležitých částí Dijkstrova algoritmu

Ve třídě *DijkstraAll* se nachází implementace Dijkstova algoritmu pro hledání nejkratší a nejrychlejší cesty. Nás ale nyní zajímá hledání nejkratší cesty, které se provádí pomocí metody *FindShortestPath*. Hlavička metody je zobrazena na následujícím kódu.

```
ResultItem[] FindShortestPath(int startNode, int endNode,bool tollWay)
```

Vstupem této metody jsou čísla počátečního a koncového uzlu a také proměnná *toolWay*, která říká, jestli chceme do vyhledávání zahrnout i placené úseky cest. Pokud je tato hodnota nastavena na true pak se placené úseky zahrnují, pokud na false pak se s nimi nepočíta. Výstupem této metody je pole obsahující prvky typu *ResultItem*. Každý z těchto prvků obsahuje informace o úseku cesty, přes který vede nalezená nejkratší cesta. Nejvíce nás ovšem z těchto informací zajímá celková délka nalezené cesty a čas strávený jízdou po ní. Ve většině případů nás bude také zajímat vykreslení nalezené cesty do mapy. Z tohoto důvodu jsou také u každé cesty také uvedeny GPS souřadnice jejího začátku a konce.

Data silniční sítě se nachází v datovém souboru na disku. Původně tyto data byly uloženy v databázi, nicméně kvůli rychlosti čtení velkého množství dat jsme byli nuceni uložit data na disk do datového souboru. Díky tomu se výrazně zlepšila rychlost načítání dat a tudíž i rychlost samotného algoritmu.

Datový soubor se skládá ze tří částí, které představují jednotlivé části silniční sítě. V souboru *File_Nodes* se nachází informace o jednotlivých uzlech silniční sítě, tedy křižovatkách. Další soubor je *File_Roads* a ten obsahuje informace o jednotlivých cestách. Posledním typem souboru je *GPSToNodeId*, který k zadaným GPS souřadnicím najde křižovatku. To slouží k rychlému nalezení identifikátoru uzlu, kde se má začít nebo skončit s hledáním. Na následujícím kódu je zobrazeno načtení těchto souborů.

```
using (File_Nodes fileNodes = new File_Nodes(location, "Nodes", false))

using (File_Roads fileRoads = new File_Roads(location, "Roads", false))

using (File_GPSToNodeId GPSToNodeId = new File_GPSToNodeId(location, "GPSToNodeId",
false)){……}
```

Při vytváření přístupu k těmto souborům je celý kód zabalen do příkazu *using*, který sám ošetřuje správné zavření souboru po ukončení jeho používání.

Dále se do konstruktoru předávají tři parametry. Prvním z nich je textový řetězec s názvem *location*, který určuje cestu, kde se datové soubory na disku nachází. Druhý parametr je název datového souboru a ten je pro každý soubor jiný. Pro přístup k uzlům je tento název Nodes, pro cesty je Roads a pro soubor s přiřazením souřadnic GPS k identifikačním číslům uzlů je název GPSToNodeId.

Posledním parametrem je pravdivostní atribut, který říká, zda se má datový soubor uložit do operační paměti, nebo se s ním bude pracovat přímo z disku. Pokud je tento atribut false pak se bude pracovat se souborem na disku, jinak při true bude soubor uložen do operační paměti. Výhoda uložení v operační paměti spočívá v rychlejším přístupu k datům, ovšem za cenu toho, že se celý datový soubor musí uložit do operační paměti, což si vyžádá více paměti RAM, než kdyby se používal soubor z disku.

Pro přístup k jednotlivým záznamům uložených v těchto datových souborech se následující část kódu.

```
Node startNode;
fileNodes.GetRow(startNode, out startNodeGPS);

Road road;
fileRoads.GetRow(RoadId, out road);
```

Pokaždé se je jako první vytvořen objekt typu *Node* či *Road*, podle toho jaký typ chceme načítat. K získání jednoho záznamu nám slouží metoda *GetRow*, která přijímá dva parametry. První parametr obsahuje proměnnou identifikačním číslem cesty či křižovatky. Druhý parametr je výstupní a říká, kam se mají načtená data uložit. Tedy do vytvořeného objektu typu *Node* nebo *Road*. Jakmile tedy máme objekty vytvořené a jsou v nich načteny hodnoty, pak můžeme přistupovat k jejich datům. V kapitole 4.1.1 jsou popsány tyto třídy podrobně i s proměnnými, které obsahují.

V kapitole 3.1.1 je popsán postup Dijsktrova algoritmu. Místo pro ukládání dat tam je popsáno jako množina navštívených a nenavštívených uzlů. V programovacím jazyku lze množinu reprezentovat kolekcí dat. Pro uchování načtených dat byla použita kolekce slovník (Dictionary), který je zobrazen v následujícím kódu.

```
Dictionary<int, NodeTemp> temp = new Dictionary<int, NodeTemp>();
```

Tento slovník má jako první hodnotu číslo, které je identifikační číslo uzlu a jako druhou hodnotu objekt typu *NodeTemp*, který obsahuje data o uzlech načtených z datového souboru a také proměnné pro uchování výpočtů algoritmu.

Další věc, kterou je potřeba vyřešit je jak od sebe rozlišit prozkoumané a neprozkoumané uzly. Z tohoto důvodu byla do třídy *NodeTemp* přidána proměnná *wasUsed*, které je pravdivostního typu a pokud je nastavena na hodnotu true, pak byl daný uzel již použit a pokud je nastavena na hodnotu false, pak tento uzel použit nebyl. Díky tomu velice rychle poznáme, zda byl uzel použit či nikoliv.

Dále je potřeba z množiny nenavštívených uzlů vybrat vždy ten s nejkratší vzdáleností od počátečního uzlu. Tuto podmínku je potřeba mít na paměti při vybírání vhodné struktury pro ukládání dat algoritmu. Tudíž je potřeba nějaká struktura, která bude sama třídit hodnoty vzdáleností podle velikosti. Opět byl pro tento účel použit slovník, ale v tomto případě se jednalo o tříděný slovník (SortedDictionary), jenž je ukázán v následujícím kódu.

```
SortedDictionary<int, List<int>> zasobnik =
                                        new SortedDictionary<int, List<int>>();
```

Tento tříděný slovník obsahuje jako první parametr hodnotu aktuální vzdálenosti a jako druhý parametr je seznam identifikačních čísel uzlů, jež mají stejnou vzdálenost od počátečního uzlu. Může totiž nastat případ, že dva uzly mají stejnou vzdálenost od počátečního uzlu. Hodnota vzdálenosti je uvedena v metrech. Třídí se podle hodnoty vzdálenosti od počátku od nejkratší po nejdelší.

V průběhu algoritmu se však nalezená vzdálenost může měnit a to tím způsobem, že se najde kratší vzdálenost od počátku. Tudíž je potřeba změnit tuto vzdálenost ve tříděném slovníku. Nynější stav to ovšem neumožňuje, protože neexistuje přímá vazba mezi identifikátorem vrcholu a vzdáleností. Šlo by procházet každou položku ve tříděném slovníku, dokud nenajdeme požadovaný uzel. Nicméně toto řešení je velice časově náročné a to hlavně při velkém množství dat. Z tohoto důvodu byl přidán další slovník, jehož kód je zobrazen níže.

```
Dictionary<int, int> NodeToCost = new Dictionary<int, int>();
```

Zobrazený slovník obsahuje slovník, který obsahuje dva parametry. První z nich představuje identifikátor uzlu a druhý vzdálenost od počátečního uzlu. Díky tomuto okamžitě víme, jakou vzdálenost má uzel se zadaným identifikátorem.

Další důležitou funkcí algoritmu je výpočet vzdálenosti od počátku v případě hledání nejkratší cesty, anebo výpočet času trvání jízdy po cestě v případě hledání nejrychlejší cesty.
Oba tyto výpočty jsou ukázány v následujícím kódu.

```
int actualCost = temp[actualNode].cost + road2.Length;
```

```
double localTime = temp[actualNode].time + (road2.Length / speed);
```

Proměnná *actualCost* se používá pro výpočet aktuální vzdálenosti nalezeného souseda uzlu. Vypočítá se jako součet vzdálenosti aktuálního uzlu od počátku a délkou cesty mezi aktuálním uzlem a jeho sousedem. Takto vypočtená vzdálenost je uvedena v metrech a provede se porovnání s hodnotou vzdálenosti v sousedovi. Pokud je nově nalezená vzdálenost menší, než původní nalezená hodnota, tak se stará hodnota přepíše novou (kratší) vzdáleností.

V případě výpočtu časové vzdálenosti je použita proměnná *localTime*, která obsahuje čas jízdy v sekundách od počátečního vrcholu. Výpočet se provede jakou součet času aktuálního uzlu od počátku a časem jízdy po cestě mezi aktuálním uzlem a jeho sousedem. Díky tomu, že známe délku cesty a její rychlostní omezení, jsme schopni vypočítat dobu jízdy po cestě. Nejprve je ale potřeba převést rychlostní omezení na rychlost v metrech za sekundu. Výsledek tohoto výpočtu je uveden v proměnné *speed*. Nyní už stačí vydělit délku cesty rychlostí speed a dostaneme čas jízdy po cestě.

Jako v předchozím případě, tak i tady je potřeba právě nalezený čas porovnávat s již nalezeným časem a pokud je nový čas kratší pak se uloží místo toho starého.

Jak už bylo řečeno, tak výstupem z algoritmu hledání cesty je pole obsahující informace o jednotlivých cestách, přes které vede nejkratší či nejrychlejší cesta. Nicméně Dijkstrův algoritmus najde pouze hodnotu nejkratší vzdálenosti, či nejkratšího času. Pro vypsání všech úseků cest, přes které vede nalezená cesta je potřeba další metoda, která se jmenuje *GetRoads*. Tato metoda samozřejmě pracuje s mezivýsledky Dijkstrova algoritmu. Využívá hlavně proměnné *fromRoad* a

fromVertex, které obsahují informace z jakého uzlu a přes jakou cestu jsme se dostali do daného uzlu. Je jasné, že metoda *GetRoads* může být zavolána až po skončení Dijkstrova algoritmu, tudíž se volá v posledním řádku Dijkstrova algoritmu a její výsledek je hned vrácen jako nalezená cesta. Kód popisující tento případ je zobrazen níže.

```
return GetRoads(ref temp, startNode, endNode, fileRoads);
```

Zobrazené metodě se předávají čtyři parametry. První z nich je slovník *temp*, který obsahuje načtená data o uzlech a k nim příslušné mezivýsledky Dijkstrova algoritmu. Další dva parametry jsou identifikační čísla počátečního a koncového vrcholu. To z toho důvodu aby se vědělo, kde cesta začíná a končí.

Poslední parametr předává přístup k souboru F*ile_Roads*, kde jsou uloženy informace o jednotlivých cestách. V průběhu algoritmu se totiž ukládají pouze identifikátory jednotlivých cest a žádné další podrobnější informace o cestách se neukládají. Výsledek této metody je vrácen Dijkstrovým algoritmem jako výsledná nalezená cesta.

5 Implementace A* algoritmu

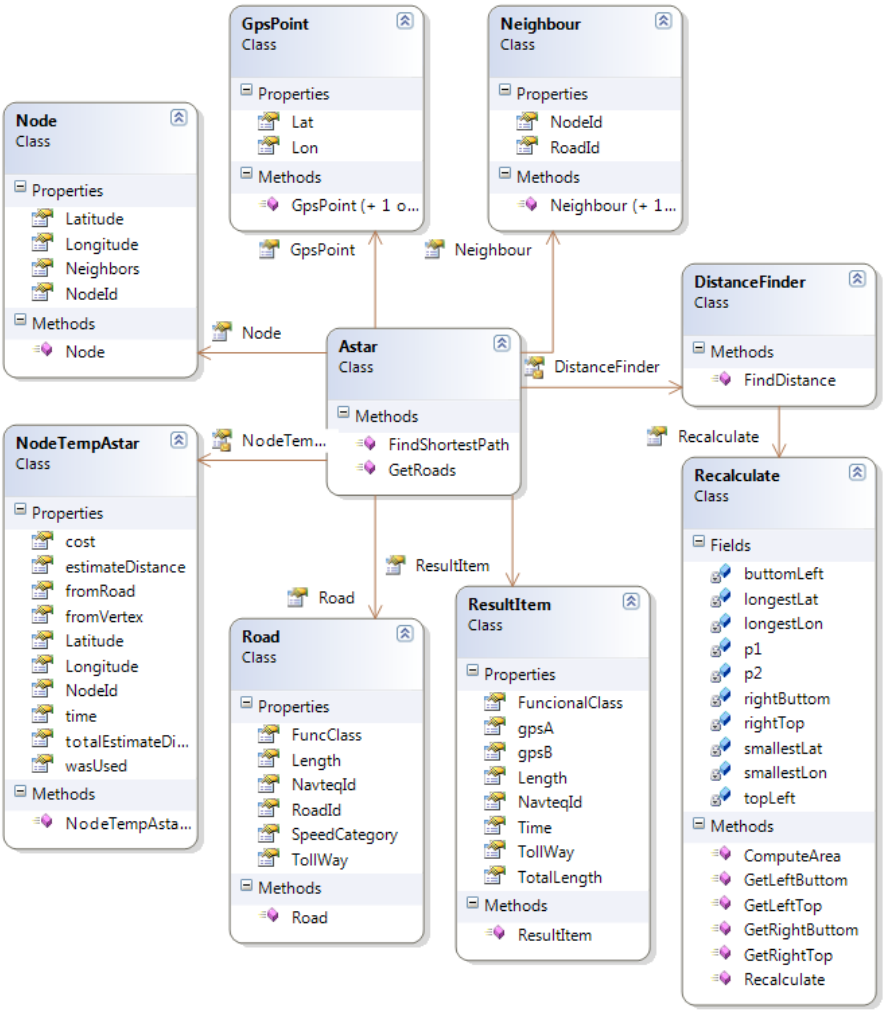
V této kapitole se budeme zabývat implementací algoritmu A*. Hlavně budou popsány všechny důležité třídy a funkce.

Na obrázku 5-1 je zobrazen třídní diagram algoritmu A*. Třída *Astar* je hlavní třída, která se stará o vyhledání nejkratší cesty. Ta se vyhledá pomocí metody *FindShortestPath*. Výsledek hledání z této metody se předá funkci *GetRoads*, která vrátí seznam cest (seznam tříd *ResultItem*), přes které vede nalezená nejkratší cesta.

5.1 Pomocné třídy A* algoritmu

Pokud porovnáme třídní diagram Dijkstrova algoritmu (obrázek 4-1) a třídní diagram A* algoritmu (obrázek 5-1), tak vidíme, že většina tříd se používá v obou algoritmech. Nové třídy oproti Dijkstrovému algoritmu jsou *NodeTempAstar*, což je třída *NodeTemp* s přidanými atributy potřebnými pro A* algoritmus, *DistanceFinder* a *Recalculate*.

Ostatní třídy jsou stejné jako v Dijkstrově algoritmu, proto nebudou dále popisovány. Informace o nich jsou uvedeny v kapitole 4.



Obrázek 5-1: Třídní diagram A* algoritmu

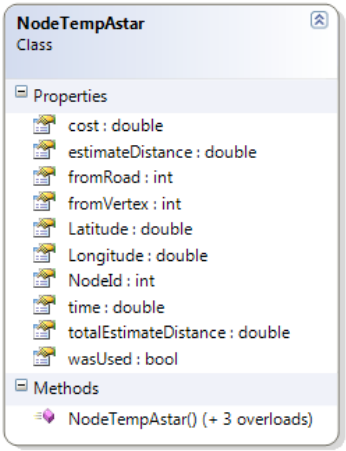
5.1.1 Třída NodeTempAstar

Jak už bylo řečeno, tak je tato třída shodná se třídou *NodeTemp*, která je popsána v kapitole 4.1.2 ovšem obsahuje navíc dva atributy. Tyto atributy se jmenují *estimateDistance* a *totalEstimateDistance* a můžeme je vidět na obrázku 4-2, kde je zobrazena celá třída *NodeTempAstar*.

Proměnná *estimateDistance* slouží k uložení odhadnuté vzdálenosti od aktuálního vrcholu do cílového vrcholu a její hodnota je uvedena v metrech. Tuto proměnnou získáme z metody *FindDistance*, která se nachází ve třídě *DistanceFinder*.

Další novou proměnou je proměnná *totalEstimateDistance*, ta se získá sečtením nalezené vzdálenosti od počátečního vrcholu do aktuálního vrcholu, čili atribut *cost* a odhadnutou vzdáleností *estimateDistance*.

Její hlavní význam spočívá v tom, že je A* algoritmus se rozhoduje právě na základě této proměnné, který další vrchol prozkoumá. Vždy je vybrán vrchol z množiny nalezených vrcholů, který má nejmenší hodnotu tohoto atributu.



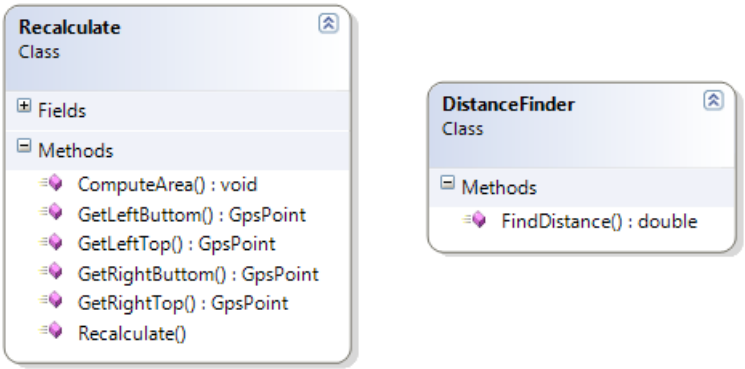
Obrázek 5-2: Třída NodeTempAstar

5.1.2 Třídy DistanceFinder a Recalculate

Tyto třídy jsou ukázány na obrázku 5-3 a jejich hlavním účelem je provést výpočet odhadu vzdálenosti mezi dvěma vrcholy. Princip tohoto výpočtu je uveden v kapitole 3.2.2.

Třída *DistanceFinder* obsahuje metodu *FindDistance*, která dostává jako parametry GPS souřadnice dvou vrcholů, mezi kterými chceme vypočítat vzdálenost. Tyto GPS souřadnice jsou následně předány objektu třídy *Recalculate*, který se stará o zjištění potřebných údajů k výpočtu vzdálenosti. Tento výpočet provede metoda *ComputeArea*.
Ostatní metody třídy *Recalculate* slouží k získání vypočtených údajů.

Jakmile jsou všechny potřebné údaje získány, tak metoda *FindDistance* vrátí hodnotu odhadované vzdálenosti, která je uvedena v metrech.



Obrázek 5-3: Třídy Recalculate a DistanceFinder

6 Optimalizace a experimenty

V této části se budeme zabývat optimalizací a experimenty s naimplementovanými algoritmy pro hledání cest.

Tato část je velice důležitá z toho důvodu, že je zapotřebí, aby směrovací algoritmy nalezly správnou (nejlepší nejkratší či nejrychlejší) cestu v krátkém čase. Rychlost vyhledání cesty, které chceme dosáhnout, je jedna sekunda v případě kratších cest a několik sekund při hledání dlouhých cest.

Další důležitou věcí, pro vytvoření vhodných experimentů, je správně zvolit místa, mezi kterými se budou vyhledávat cesty. Je potřeba aby se otestovaly všechny možné typy případů hledání cest. Takže experimenty musí obsahovat hledání krátkých cest, dlouhých cest a cest vedoucí od jednoho konce republiky k druhému.

Jednotlivá místa, mezi kterými se budou hledat cesty, jsou uvedeny v tabulce 6-1. Ke každému místu jsou uvedeny jeho GPS souřadnice, pro určení jeho přesné pozice. V této tabulce jsou obsaženy všechny typy cest. Mezi krátké patří cesta mezi Ostravou a Karvinou, mezi dlouhé patří například cesta mezi Prahou a Ostravou. Cesta vedoucí od jednoho konce republiky ke druhému je cesta mezi Aší a Jablunkovem a cesta mezi Českými Budějovicemi a Libercem.

Kvůli tomu, aby výsledky experimentů byli co nejpřesnější a dali se jednoduše porovnávat, tak se při každém hledání použije stejná výchozí či cílová GPS pozice daného místa. Jak už bylo řečeno, tak tato GPS pozice je zobrazena v tabulce 6-1. V dalších částech kdy se bude pracovat s těmito místy, už u nich nebude zobrazena jejich GPS pozice. Proto, bude-li zapotřebí zjistit přesnou polohu těchto míst, tak se bude zapotřebí podívat do této tabulky.

Místo	Latitude	Longitude
Aš	50,30882	12,14472
Brno	49,19821	16,60733
České Budějovice	48,97793	14,47252
Hradec Králové	50,21124	15,81638
Jablunkov	49,55267	18,85210
Karviná	49,83865	18,54371
Liberec	50,77075	15,06174
Ostrava	49,83370	18,16921
Praha	50,11764	14,45255

Tabulka 6-1: Místa určená pro hledání cesty

6.1 Optimalizace zdroje dat

Ke správné a rychlé funkci algoritmů je zapotřebí mít pouze k dispozici pouze potřebná data pro tyto algoritmy a zajistit k nim rychlý přístup. Z tohoto důvodu bylo zapotřebí dobře optimalizovat zdroj dat.

K dispozici jsme měli databázi od společnosti NAVTEQ, které obsahovala kompletní informace o silniční síti České republiky. Ta obsahuje 1 329 689 uzlů a 1 571 500 cest. Nicméně tato databáze obsahovala velké množství, pro směrování, nepotřebných údajů. Z tohoto důvodu jsme přesunuli pouze potřebná data do nové databáze a tu jsme použily pro zdroj dat k algoritmům.

Toto řešení nebylo zcela ideální, kvůli časům vyhledávání algoritmů. Hledání cesty Dijkstrovým algoritmem z Ostravy do Karviné trvalo okolo minuty, což je velice dlouhá doba, vezmeme-li v úvahu jak blízko sobě jsou obě místa.

Z tohoto důvodu jsme se rozhodli data uložit jako binární soubory na disk. V těchto souborech jsou uloženy objekty typu *Node* a *Road*, které obsahují informace o cestách a uzlech. Třídy těchto objektů jsou zobrazeny na obrázku 4-2. Pro každý typ objektu je použit samostatný binární soubor. Ten se skládá ze dvou seznamů, z nichž první obsahuje setříděné identifikátory objektů a také odkaz do druhého seznamu, v němž se nachází objekty. Druhý soubor, má tu zvláštnost, že číslo identifikátoru objektu odpovídá číslu řádku, na kterém se nachází. Díky tomu je možné okamžitě přistoupit k jakémukoliv objektu.

Abychom dosáhli ještě rychlejšího přístupu k datům, tak je možné určit, při vytváření přístupu k datům, jestli se mají nahrát do operační paměti počítače. Tudíž čtení dat bude probíhat z operační paměti, která je rychlejší, oproti čtení z pevného disku. Uložení do operační paměti je možné hlavně proto, že velikost binárních souborů s daty je poměrně malá. Celková velikost těchto binárních souborů pro Českou republiku je 105 MB.

Díky těmto binárním souborům uložený v operační paměti, jsme schopni dát vyhledávacím algoritmům zdroj dat s rychlým přístupem a velkou rychlostí čtení.

6.2 Optimalizace Dijkstrova Algoritmu

Tato část se zabývá optimalizací Dijkstrova algoritmu. Jsou zde ukázány experimenty, které mají za cíl, aby čas nalezení cesty byl co nejkratší.

V první části jsou ukázány výsledky hledání cest, které jsou vráceny neoptimalizovaným Dijkstrovým algoritmem. Dále pak budou vysvětleny a ukázány jednotlivé optimalizační kroky a výsledky hledání cest při použití těchto optimalizací. Pro každou cestu, která se má najít se hledá nejkratší a nejrychlejší cesta. To je z důvodu vzájemného porovnání výsledků a přesnějšího provedení optimalizací.

6.2.1 Hledání cest neoptimalizovaným Dijkstrovým algoritmem

Výsledky hledání nejkratších a nejrychlejších cest neoptimalizovaným Dijkstrovým algoritmem jsou uvedeny v tabulkách 6-2 a 6-3.

Nalezené nejkratší cesty jsou uvedeny v tabulce 6-2. Pokud se podíváme na čas vyhledávání cesty tak, zjistíme, že nejdelší vyhledávání se pohybuje okolo deseti sekund. Tento čas trvá vyhledání cesty z Aše do Jablunkova a taky se mu blíží čas hledání cesty z Prahy do Ostravy.

Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,54s	31 412m	34m	61 772
Praha-Ostrava	9,6s	311 168m	4h 52m	1 250 716
Aš-Jablunkov	10,2s	562 568m	7h 42m	1 329 298
Č. Budějovice-Liberec	7,8s	233 924m	3h 4m	1 017 586
Brno-Hradec Králové	3,5s	140 785m	1h 56m	502 741

Tabulka 6-2: Hledání nejkratších cest

Vzhledem k tomu, že cesta z Aše do Jablunkova patří mezi nejdelší v celé republice, tak můžeme tvrdit, že vyhledání jakékoliv nejkratší cesty bude provedeno trvat nejvýše 10 sekund.

Nejkratší cesta, která byla hledána, mezi Ostravou a Karvinou byla nalezena za půl vteřiny. Tento čas se dal očekávat vzhledem k tomu, jak blízko k sobě tyto místa jsou.

V tabulce 6-3 jsou nalezené nejrychlejší cesty. Na první pohled jde vidět, že pokud porovnáme nalezené časy nejkratších a nejrychlejších cest, tak vyhledání nejrychlejších cest je více časově náročné. To je způsobeno rozdílným způsobem výpočtu metriky algoritmu a také tím, že jsou náročnější mezi-výpočty u hledání nejrychlejší cesty.

I v tomto případě má nejdelší čas vyhledání nejdelší možná cesta z Aše do Jablunkova. Nyní ovšem čas hledání vzrostl až na 16 sekund. Je zajímavé, že tohoto času dosáhla i nejrychlejší cesta mezi Prahou a Ostravou, která je ovšem 282 kilometrů kratší. Nicméně pokud se díváme na počet prozkoumaných vrcholů, tak vidíme, že v obou případech byl prozkoumán skoro stejný počet vrcholů. Z toho vyplývá, že bude existovat závislost mezi časem nalezení cesty a počtem vrcholů, které se přitom prohledají.

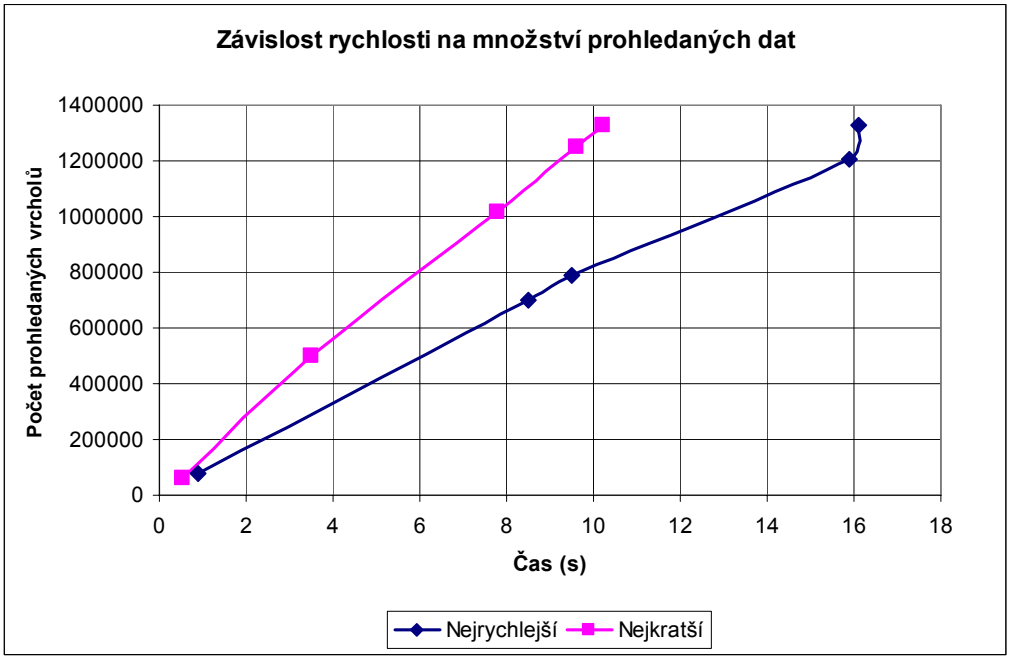
Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,9s	34 781m	31m	77 046
Praha-Ostrava	15,9s	368 486m	3h	1 205 605
Aš-Jablunkov	16,1s	650 618m	6h 5m	1 328 854
Č. Budějovice-Liberec	9,5s	250 114m	2h 33m	787 172
Brno-Hradec Králové	8,5s	141 900m	1h 52m	698 566

Tabulka 6-3: Hledání nejrychlejších cest

Na grafu 6-1 je vidět závislost mezi rychlostí algoritmu a množstvím prohledaných vrcholů. V obou případech, tj. při hledání nejkratší a nejrychlejší cesty, je přímá úměra mezi rychlostí algoritmu a množstvím prohledaných vrcholů.

Z této závislosti je potřeba dále vycházet, protože další optimalizace by se měla snažit určitým způsobem zmenšit množinu dat, nad kterou se bude vyhledávat.

Samotný Dijkstrův algoritmus, tak jak je naimplementován, již nelze žádným jiným způsobem urychlit. Právě proto je jediná další možnost optimalizace zredukovat množinu prohledávaných dat. Nicméně při redukci musíme být opatrní, ať jsou zredukována pouze data, která jsou pro hledání cesty nepotřebná.



Graf 6-1: Závislost rychlosti hledání na množství prohledaných vrcholů

6.2.2 Optimalizace oříznutím množiny vyhledávání

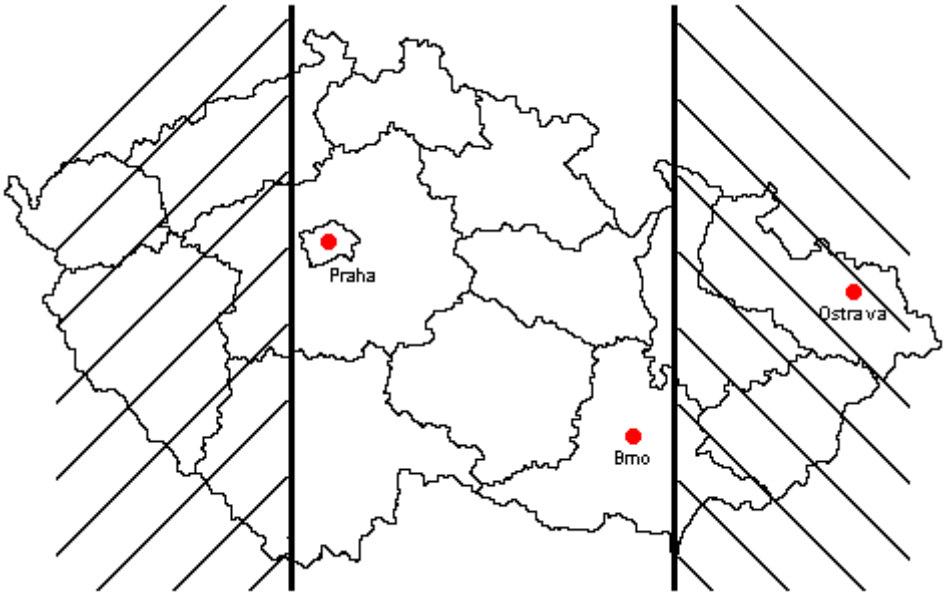
Na obrázku 6-1 je zobrazen princip optimalizace pomocí oříznutí množiny vyhledávání. V tomto případě se provádí ořez pro hledání cesty mezi Brnem a Prahou.

Z principu fungování Dijkstrova algoritmu víme, že prochází postupně celou množinu dat a vybírá vždy vrchol s nejkratší vzdáleností od počátku. Tudíž prochází data jakoby ve zvětšujících se vlnách od počátečního vrcholu.

Pokud tedy například hledáme cestu z Brna do Prahy, tak ještě než je nalezena cesta do Prahy, tak najde algoritmus cestu do Ostravy. To je způsobeno tím, že Ostrava leží blíže k Brnu než Praha. Algoritmus ji tedy díky tomu, že vybírá vždy nejkratší vzdálenost od počátečního vrcholu, najde dřív.

Hlavní myšlenka této optimalizace tedy spočívá v oříznutí množiny dat, které leží na opačné straně, než leží cílový vrchol. Je tedy nutné vypočítat směr od počátečního vrcholu k cílovému vrcholu. Nicméně nemůžeme oříznout množinu, která sice leží na opačné straně, než se bude vyhledávat, ale je v blízkosti počátečního vrcholu. Je zapotřebí dát menší rezervu na opačnou stranu, než se bude vyhledávat. Jako tato rezerva byla zvolena vzdálenost 10 kilometrů. Ořez se tedy provede na opačné straně vyhledávání ve vzdálenosti 10 kilometrů od počátečního a koncového vrcholu.

Díky tomu se zmenší vyhledávací množina, a tudíž se najde hledaná cesta rychleji. Nyní tedy při hledání cesty z Brna do Prahy se již nebudou prohledávat data o Ostravě, či jiné data ležící mimo hledanou cestu.



Obrázek 6-1: Princip ořízнutí dat při hledání cesty Brno-Praha

Výsledky vyhledání nejkratších cest jsou uvedeny v tabulce 6-4. Pokud tyto výsledky porovnáme s hledáním nejkratších cest neoptimalizovaným algoritmem (tabulka 6-2), tak vidíme, že nalezené cesty mají stejnou délku, čili jsou shodné.

Co se však poměrně razantně změnilo je čas hledání a počet prohledaných vrcholů. Vyhledání cesty z Prahy do Ostravy se zrychlilo o 2,9 sekundy na 6,7 sekund. To je způsobeno snížením prohledaných vrcholů o 361 000.

Vůbec největší změnu zaznamenalo hledání cesty z Českých Budějovic do Liberce, kde se čas hledání snížil o neuvěřitelných 6 sekund na 1,8 sekundy. Množství prohledaných vrcholů se snížilo o 757 000.

Pokud se ovšem podíváme na hledání cesty přes celou republiku z Aše do Jablunkova, tak nedošlo k žádnému snížení. To se dalo očekávat, protože obě místa leží na okraji republiky, kde není prakticky co oříznout.

Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,45s	31 412m	34m	50 900
Praha-Ostrava	6,7s	311 168m	4h 52m	889 020
Aš-Jablunkov	10,1s	562 568m	7h 42m	1 329 298
Č. Budějovice-Liberec	1,8s	233 924m	3h 4m	260 579
Brno-Hradec Králové	1,6s	140 785m	1h 56m	208 721

Tabulka 6-4: Hledání nejkratších cest s ořízнutím

V tabulce 6-5 jsou zobrazeny výsledky vyhledání nejrychlejších cest. Pokud ji opět porovnáme s tabulkou 6-3, tak dojdeme zase k tomu, že byly nalezeny úplně stejné cesty jako při hledání neoptimalizovaného algoritmu.

Největšího snížení rychlosti vyhledávání je opět dosaženo při hledání cesty mezi Českými Budějovicemi a Libercem, kde je dosaženo snížení rychlosti vyhledávání o 6,9 sekund na 2,6 sekund a mezi Brnem a Hradcem Králové, kde došlo ke snížení rychlosti vyhledávání o 6,3 sekund na 2,2 sekund.

Co se týče hledání cesty mezi Aší a Jablunkovem, tak v tomto případě opět nedošlo k žádné změně rychlosti.

Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,7s	34 781m	31m	56 573
Praha-Ostrava	10s	368 486m	3h	843 909
Aš-Jablunkov	15,9s	650 618m	6h 5m	1 328 854
Č. Budějovice-Liberec	2,6s	250 114m	2h 33m	240 118
Brno-Hradec Králové	2,2s	141 900m	1h 52m	203 791

Tabulka 6-5: Hledání nejrychlejších cest s ořízntím

6.2.3 Optimalizace vybráním vhodných dat

Optimalizace ořízntím množiny vyhledávání přinesla velké zlepšení rychlosti vyhledávání cesty pro cesty, které nevedou přes celou republiku. Pro cestu z Aše do Jablunkova však tato optimalizace neznamenala žádnou změnu ve zlepšení rychlosti vyhledávání. Z tohoto důvodu je zapotřebí dále optimalizovat vyhledávání.

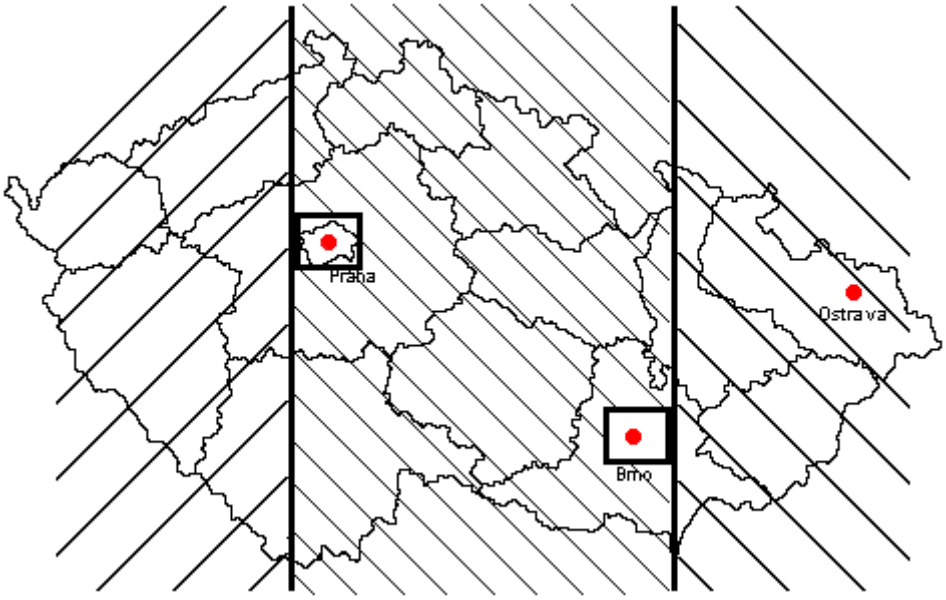
Nicméně je vidět, že odstranění nepotřebných dat má příznivé účinky pro zrychlení vyhledávání. Takže jako další optimalizace bylo zvoleno odstranění nepotřebných dat v množině vyhledávání.

V předchozí optimalizaci se odstranily data, které byly pro hledání zbytečné, nicméně v této optimalizaci se z množiny dat, které jsou potřebná pro nalezení cesty, vyberou jen ty data, která jsou opravdu zapotřebí.

Jako příklad může posloužit, že při hledání nejrychlejší cesty je zbytečné procházet cesty s nízkým rychlostním limitem, kterých bude v množině vyhledávání poměrně hodně. Tím se zmenší množina pro vyhledávání a zrychlí se tím hledání cesty.

Rozhodnutí, která cesta je pomalá a která ne, můžeme provést díky atributu *FuncClass*, který je uveden u každé cesty. Tento atribut má hodnoty v rozsahu 1 až 5, přičemž hodnota 1 značí nejrychlejší cestu a hodnota 5 značí nejpomalejší cestu.

Princip této optimalizace je zobrazen na obrázku 6-2. Opět je zde uveden příklad na hledání cesty mezi Brnem a Prahou. Kolem počátečního a koncového místa se v okruhu deseti kilometrů vybírají všechny cesty. V ostatních částech vyhledávací množiny se používají pouze cesty, které mají vhodnou rychlost. Tudíž se provede odstranění těch nejpomalejších cest. Tímto způsobem jsme schopni opět vhodně zredukovat množinu vyhledávání.



Obrázek 6-2: Princip vybrání vhodných dat při hledání cesty Brno-Praha

Výsledky hledání nejkratších cest jsou zobrazeny v tabulce 6-6. Při hledání se nepoužily cesty, které měli hodnotu atributu FuncClass rovnu pěti. Tudíž se nepoužily ty úplně nejpomalejší cesty.

Díky tomu došlo k výraznému snížení času hledání hlavně u cest vedoucích přes větší část republiky. Pokud se podíváme na nejdelší cestu vedoucí z Aše do Jablunkova, tak vidíme, že došlo k snížení doby hledání z 10,1 sekund na 2,3 sekundy, což je velmi dobrá rychlost vyhledání cesty. K tak výraznému snížení došlo hlavně proto, že byla snížena množina prohledávaných dat z 1 328 854 na 315 683. Nicméně celková nalezená cesta je o 6,6 kilometrů horší, než v předchozí optimalizaci. Vzhledem k tomu, jak jsou od sebe místa vzdálená je tento rozdíl prakticky zanedbatelný. Pokud navíc vezmeme v úvahu, že nalezená cesta nevede přes žádnou cestu, která je velmi pomalá, či je ve špatném stavu, a tudíž by mohl být problém se přes ni dostat, pak to můžeme brát jako výhodu.

To platí i pro ostatní cesty, které mají minimální rozdíl v nalezené vzdálenosti oproti předchozí optimalizaci.

Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,4s	31 764m	34m	40 076
Praha-Ostrava	2,3s	317 944m	4h 27m	313 436
Aš-Jablunkov	2,3s	569 181m	7h 26m	315 683
Č. Budějovice-Liberec	1,4s	235 704m	3h 1m	196 781
Brno-Hradec Králové	1,5s	140 785m	1h 56m	208 721

Tabulka 6-6: Hledání nejkratších cest s vybráním vhodných dat

Při hledání nejrychlejších cest se nepoužily cesty, které měli hodnotu atributu *FuncClass* rovnu čtyřem nebo pěti. Výsledky hledání jsou zobrazeny v tabulce 6-7.

Pokud porovnáme výsledky s předchozí optimalizací hledání nejrychlejší cesty, tak vidíme, že až na cestu z Prahy do Ostravy jsou všechny délky cest i časů cest stejné. Z toho lze usoudit, že tato optimalizace je velice dobrá.

Největší snížení času, při stejném výsledku hledání, je dosaženo při hledání cesty z Aše do Jablunkova, kde jsme se dostali z 15.9 sekund na 1,3 sekundy. To je více, než dvanáctinásobné zrychlení oproti původnímu hledání. Množina dat, ve které se vyhledávalo, se zmenšila o 1,2 miliónu na 127 571.

Cesta	Čas hledání	Délka cesty	Čas cesty	Prohledané vrcholy
Ostrava-Karviná	0,5s	34 781m	31m	42 065
Praha-Ostrava	2,2s	376 501m	3h 15m	211 908
Aš-Jablunkov	1,3s	650 618m	6h 5m	127 571
Č. Budějovice-Liberec	1,8s	250 114m	2h 33m	169 215
Brno-Hradec Králové	2,2s	141 900m	1h 52m	203 791

Tabulka 6-7: Hledání nejrychlejších cest s vybráním vhodných dat

Díky tomu, že použijeme obě optimalizace dat (optimalizaci ořízнutím množiny a optimalizaci výběru vhodných dat), jsme schopni najít jakoukoliv nejkratší či nejrychlejší cestu v čase pod 2,3 sekundy.

Tato rychlost splňuje očekávání, které jsme si stanovili na začátku experimentů. To je hlavní důvod, proč se s touto rychlostí vyhledávání spokojíme. Další důvod je ten, že již neexistuje jiný způsob jak více ořezat množinu prohledávaných dat, tak aby nedošlo k výraznému zhoršení délky či času nalezené cesty.

6.3 Optimalizace A* algoritmu

Klíčová věc, na které A* algoritmus stojí či padá, je správné odhadnutí vzdálenosti mezi dvěma vrcholy. Celá tato kapitola se zabývá právě touto problematikou.

Experimenty budou prováděny při hledání cesty mezi Prahou a Ostravou a také při hledání cesty mezi Aší a Jablunkovem. Tyto cesty byly zvoleny hlavně proto, že vedou přes větší část republiky. Tudíž se ukáže fungování a rychlost algoritmu na velké množině dat.

První věc, která nás napadne, pokud budeme chtít určit vzdálenost po silnici mezi dvěma vrcholy je vypočítat přímou (leteckou) vzdálenost mezi nimi. Tato vzdálenost sice není přesná skutečná vzdálenost po silnici mezi dvěma vrcholy, ale blíží se jí a také má tu vlastnost, že nám odhad vzdálenosti omezuje zdola. Díky ní tedy víme, jaká je nejkratší možná vzdálenost mezi dvěma vrcholy. Nemůže totiž nastat případ, kdyby skutečná cesta mezi vrcholy byla kratší, než letecká vzdálenost mezi nimi.

Ve většině případů bude tedy skutečná vzdálenost mezi dvěma vrcholy vždy o něco větší než vypočtená letecká vzdálenost. Proto je potřeba nalézt vhodný poměr mezi leteckou a skutečnou vzdáleností. Z tohoto důvodu je provedeno několik hledání cest s různým nastavením odhadu vzdálenosti.

Pro různé nastavení vzdálenosti je použita konstanta, kterou se vynásobí zjištěná letecká vzdálenost. Ta zajišťuje již zmíněný poměr mezi skutečnou a leteckou vzdáleností. Pokud tedy máme dvě letecké vzdálenosti o hodnotách 100 a 20 kilometrů a konstantu o hodnotě 1,4, pak bude odhadnutá skutečná vzdálenost mít hodnotu 140 a 28 kilometrů. Tyto vzdálenosti se zcela jistě blíží skutečné vzdálenosti více, než vypočtená letecká vzdálenost.

Hlavní výhoda tohoto postupu je, že zachovává poměr mezi dlouhými a krátkými vzdálenostmi. Každá vzdálenost se zvýší jen o určitý násobek své velikosti. Pokud tedy máme například konstantu 1,4, pak se zvýší vzdálenost všech vypočtených vzdáleností o 40% své původní velikosti. Pokud bychom místo násobení konstanty použili přičítání určité vzdálenosti, pak by to nemělo žádný vliv, protože by se všechny vzdálenosti zvýšily o stejnou hodnotu.

V tabulce 6-8 jsou zobrazeny výsledky hledání cesty z Prahy do Ostravy při použití různých konstant. U každé cesty je uvedena použitá konstanta, čas hledání délka nalezené cesty a počet pohledaných vrcholů. Dále je u každé nalezené cesty uvedeno její porovnání se vzdáleností nalezenou Dijkstrovým algoritmem, a to jak tím bez žádných optimalizací, ten je pojmenován klasický a tím optimalizovaným. Díky tomu můžeme porovnávat, jak moc se nalezená cesta liší oproti nejkratší cestě.

Pokud se tedy podíváme na tuto tabulku, tak vidíme, že pokud je konstanta rovna jedné, což znamená, že používáme leteckou vzdálenost, pak je vidět, že najdeme cestu, která je naprosto shodná s neoptimalizovaným Dijkstrovým algoritmem. Z toho lze usuzovat, že pokud počítáme s leteckou vzdáleností, pak se A* algoritmus chová jako Dijkstrův algoritmus. Nicméně tento stav není žádoucí, vzhledem k tomu, že bylo prozkoumáno obrovské množství vrcholů k nalezení cesty.

Pokud dále změníme konstantu z 1 na 1,05 tak vidíme, že se zmenšila množina prozkoumaných vrcholů o 70 tisíc a přitom se našla cesta, která je delší jen o tři metry než nejkratší cesta. Nyní už můžeme pozorovat, že se výsledky začínají přibližovat tomu, co očekáváme od A* algoritmu, čili to že se zmenšuje množina prohledaných vrcholů a délka nalezené cesty se blíží délce nejkratší cesty.

Takto můžeme postupovat dále. Je vidět, že s každým zvětšením konstanty se zvětší i rozdíl délky cesty oproti Dijkstovému algoritmu. Nicméně taky se zkrátí čas hledání a počet prozkoumaných vrcholů.

				Rozdíl oproti Dijkstovi	
Konstanta	**Čas hledání**	**Délka cesty**	**Prohledané vrcholy**	**Klasický**	**Optimalizovaný**
1	4,4s	311 168m	403 961	0m	-6 776m
1,05	3,7s	311 171m	331 254	3m	-6 773m
1,1	3,1s	314 090m	272 917	2 922m	-3 854m
1,15	2,3s	315 561m	199 798	4 393m	-2 383m
1,2	1,2s	315 969m	107 741	4 801m	-1 975m
1,25	0,5s	317 062m	43 365	5 894m	-882m
1,3	0,3s	318 316m	25 430	7 148m	372m
1,35	0,2s	319 175m	11 067	8 007m	1 231m
1,4	0,16s	325 640m	7 055	14 472m	7 696m
1,5	0,15s	328 696m	5 727	17 528m	10 752m
1,6	0,12s	328 998m	4 809	17 830m	11 054m
2	0,16s	342 572m	3 564	31 404m	24 628m
3	0,1s	355 435m	2 991	44 267m	37 491m

Tabulka 6-8: Hledání cesty z Prahy do Ostravy s různými konstantami

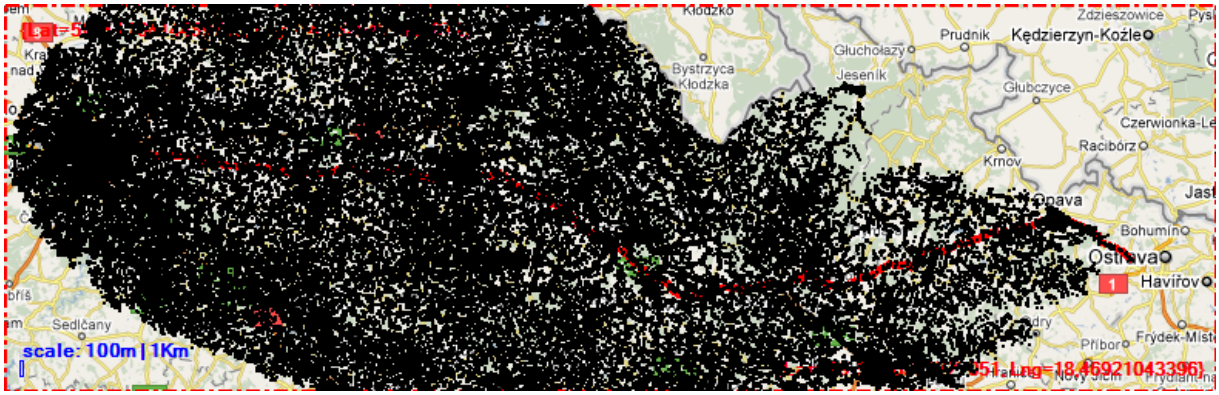
Pokud vezmeme případ s konstantou 1,5, pak jsme schopni nalézt cestu za 0,15 sekundy s použitím necelých šesti tisíc vrcholů, při nalezení cesty o 17,5 kilometrů delší, než je nejkratší cesta. Když toto číslo převedeme na procenta, tak při použití konstanty 1,5 najdeme cestu pouze o 5% delší než je nejkratší cesta. Což není vůbec špatné, vzhledem k tomu, že byl použit pouze zlomek vrcholů pro hledání v porovnání s Dijkstrovým algoritmem.

Konstanty větší než 1,6 už dávají poměrně velkou odchylku vzdáleností a přitom počet prohledaných vrcholů a čas hledání cesty je podobný jako, když jsou použity konstanty 1,5 a 1,6. Z toho to důvodu jsou konstanty s hodnotou 2 a více nevhodné pro použití při hledání cesty.

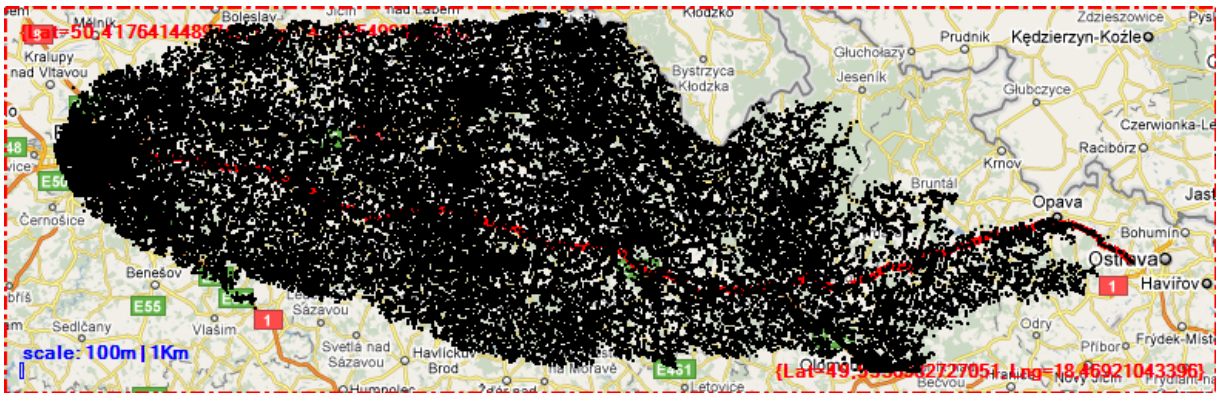
K tomu abychom lépe pochopili A* algoritmus nám pomůže zobrazení, jak konstanta ovlivňuje množství prohledávaných dat. To je zobrazeno na obrázcích 6-3 až 6-7.

Na obrázku 6-3 vidíme hledání cesty s konstantou 1. Jak je vidět, tak se prochází většina vrcholů na cestě mezi Prahou a Ostravou. V tomto případě se chová A* algoritmus jako Dijkstrův algoritmu. Nicméně pokud změníme konstantu na 1,1, což je zobrazeno na obrázku 6-4, tak vidíme, že dochází ke zmenšení množiny prohledaných vrcholů. Množina prohledaných vrcholů se zúží a začne se přibližovat nalezené cestě. Tento postup se opakuje pro konstantu 1,2 a 1,3, které jsou zobrazeny na obrázcích 6-5 a 6-6.

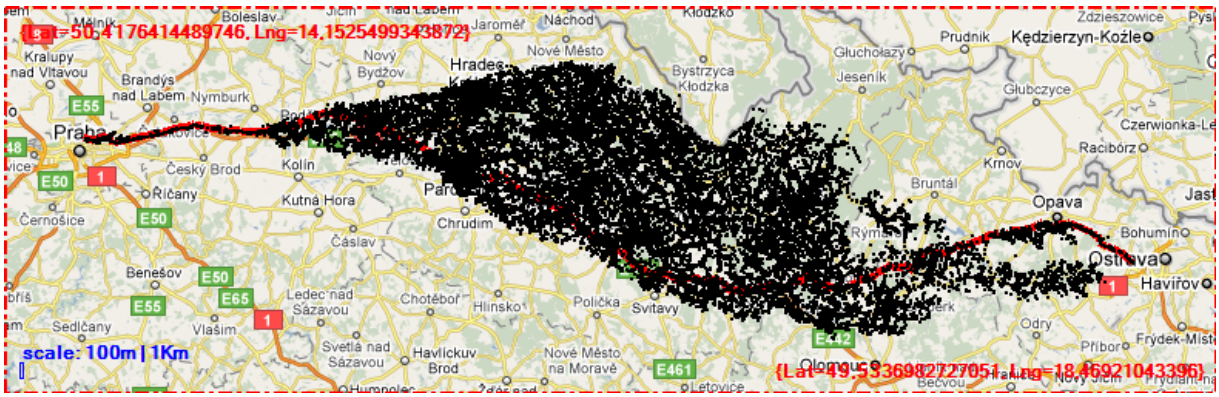
Při použití konstanty o hodnotě 1,4, která je zobrazena na obrázku 6-7, je vidět, že většina prozkoumaných vrcholů již leží na nalezené cestě.



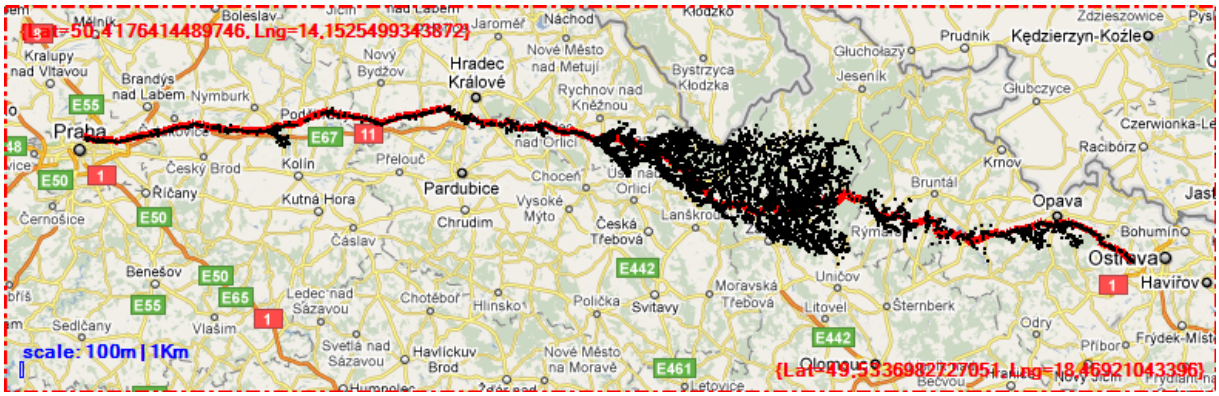
Obrázek 6-3: Prohledané vrcholy při hledání cesty Praha-Ostrava s konstantou 1



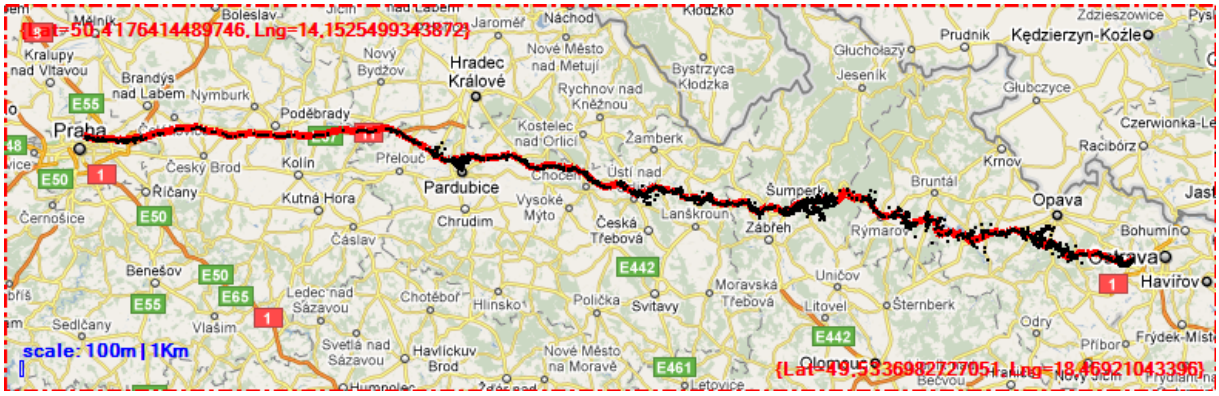
Obrázek 6-4: Prohledané vrcholy při hledání cesty Praha-Ostrava s konstantou 1,1



Obrázek 6-5: Prohledané vrcholy při hledání cesty Praha-Ostrava s konstantou 1,2



Obrázek 6-6: Prohledané vrcholy při hledání cesty Praha-Ostrava s konstantou 1,3



Obrázek 6-7: Prohledané vrcholy při hledání cesty Praha-Ostrava s konstantou 1,4

Nyní když už víme, jak jsou vrcholy prohledávány A* algoritmem v závislosti na konstantě, tak můžeme přejít k dalším experimentům s konstantou při hledání cesty z Aše do Jablunkova. Výsledky hledání této cesty jsou uvedeny v tabulce 6-9.

Výsledky hledání jsou principiálně podobné předchozímu hledání cesty. Opět, při použití konstanty 1 se A* algoritmus chová jako Dijkstrův algoritmus a najde stejnou nejkratší cestu. Při použití konstanty 1,05 je nalezená cesta o pouhých 12 metrů delší, než nejkratší cesta, nicméně došlo ke snížení množiny prohledaných vrcholů o 200 000.

Každé další zvýšení konstanty sebou nese zkrácení času hledání, zmenšení množiny prohledaných vrcholů a zvětšení délky cesty oproti nejkratší nalezené cestě.

Do maximálního 5 % rozdílu délky nalezené cesty v porovnání s nejkratší cestou se v tomto případě dostala cesta při použití konstanty 1,35.

				Rozdíl oproti Dijkstovi	
Konstanta	**Čas hledání**	**Délka cesty**	**Prohledané vrcholy**	**Klasický**	**Optimalizovaný**
1	11,2s	562 568m	918 337	0m	-6 613m
1,05	8,8s	562 580m	722 536	12m	-6 601m
1,1	5s	562 810m	411 957	242m	-6 371m
1,15	1,4s	563 539m	124 061	971m	-5 642m
1,2	0,5s	570 964m	46 355	8 396m	1 783m
1,25	0,4s	578 426m	31 701	15 858m	9 245m
1,3	0,4s	589 434m	27 337	26 866m	20 253m
1,35	0,2s	591 126m	13 357	28 558m	21 945m
1,4	0,2s	598 131m	13 651	35 563m	28 950m
1,5	0,2s	612 403m	12 133	49 835m	43 222m

Tabulka 6-9: Hledání cesty z Aše do Jablunkova s různými konstantami

Díky těmto předchozím dvěma experimentům s hledáním cesty při různých konstantách jsme si udělali představu o tom, k jakým výsledkům vede použití jednotlivých konstant.

Je vidět, že A* algoritmus je velice dobře škálovatelný. Lze totiž vhodným nastavením konstanty vyhledat velmi rychle nejkratší cestu i nad velkou množinou dat.

Bylo zjištěno, že při použití konstanty o hodnotě 1 se A* algoritmus chová jako Dijkstrův algoritmus, prohledává všechny vrcholy mezi počátečním a koncovým vrcholem, což má za následek dlouhý čas hledání cesty. Z tohoto důvodu není vhodné tuto konstantu používat.

Vzhledem k tomu, že jsme měli k dispozici data silniční sítě pouze České republiky, tak nebylo možné provádět experimenty při hledání cest přes celou Evropu. Což je vzhledem k dobré škálovatelnosti algoritmu A* škoda.

Při hledání cesty v České republice je tedy podle experimentů nejlepší použít konstanty v rozmezí kolem 1,15 a 1,2. Výsledky hledání cest s těmito konstantami jsou uvedeny v tabulkách 6-10 a 6-11.

Co se týče hledání cesty na území Evropy, tak v tom případě bude zapotřebí použít konstantu v rozmezí 1,3 až 1,4. Sice se jedná jen o odhad, vzhledem k tomu, že nemáme potřebná data, ale předchozí experimenty tomu nasvědčují.

				Rozdíl oproti Dijkstovi	
Cesta	Čas hledání	Délka cesty	Prohledané vrcholy	Klasický	Optimalizovaný
Ostrava-Karviná	0,15s	31 464m	6 084	52m	-300m
Praha-Ostrava	2,3s	315 561m	199 798	4 393m	-2 383m
Aš-Jablunkov	1,4s	563 539m	124 061	971m	-5 642m
Č. Budějovice-Liberec	0,5s	235 042m	31 738	1 118m	-662m
Brno-Hradec Králové	0,4s	141 203m	32 591	418m	418m

Tabulka 6-10: Hledání cest A* algoritmem s konstantou 1,15

				Rozdíl oproti Dijkstovi	
Cesta	Čas hledání	Délka cesty	Prohledané vrcholy	Klasický	Optimalizovaný
Ostrava-Karviná	0,14s	31 879m	4 708	467m	115m
Praha-Ostrava	1,2s	315 969m	107 741	4 801m	-1 975m
Aš-Jablunkov	0,5s	570 964m	46 355	8 396m	1 783m
Č. Budějovice-Liberec	0,2s	235 656m	13 539	1 732m	-48m
Brno-Hradec Králové	0,2s	147 338m	10 193	6 553m	6 553m

Tabulka 6-11: Hledání cest A* algoritmem s konstantou 1,2

6.4 Porovnání výsledků a zhodnocení algoritmů

V této části se budeme zabývat porovnáním výsledků všech naimplementovaných algoritmů. Dále budou tyto výsledky porovnány s výsledky získanými z webové aplikace hledání cest, která se nachází na adresy mapy.cz. Díky tomuto srovnání udivíme, jak moc se výsledky hledání cest liší od profesionální aplikace, která je určena k tomuto účelu.

Pro porovnání výsledků Dijkstrova algoritmu jsou uvedeny obě jeho implementované verze, jak klasická (neoptimalizovaná), tak i optimalizovaná. Hlavně nás ovšem budou zajímat výsledky z optimalizované verze, která najde cestu v podstatně kratším čase, než verze neoptimalizovaná. Hledání cest s algoritmem A* bude prováděno s použití konstant o hodnotách 1,15 a 1,2. Tyto hodnoty jsou vhodné pro hledání cest v České republice.

V tabulce 6-12 jsou uvedeny výsledky hledání nejkratších cest. Pokud se podíváme na obě verze Dijkstrova algoritmu, tak vidíme, že nalezené cesty jsou vždy o něco kratší, než cesty nalezené přes mapy.cz. Optimalizovaný Dijkstrův algoritmus dává velmi podobné výsledky jako mapy.cz. V nejhorším případě se liší o pouhé 2 kilometry.

Výsledky z A* algoritmu mají taktéž, až na jednu výjimku, kratší délky nalezených cest oproti cestám nalezeným přes mapy.cz. Tato výjimka je při hledání cesty z Brna do Hradce Králové s konstantou 1,2, kde je nalezená cesta o 5 kilometrů delší.

Pokud porovnáme optimalizovaný Dijkstrův algoritmus s A* algoritmem, tak nelze jednoznačně určit, který algoritmus dává lepší výsledky. V některých případech, jako například hledání cesty Ostrava-Karviná jsou výsledky prakticky shodné, jindy, například při hledání cesty Praha-Ostrava dává lepší výsledky A* algoritmus a v jiných případech, jako například hledání cesty Brno-Hradec Králové dává lepší výsledky optimalizovaný Dijkstrův algoritmus.

		Dijkster		A*	
Cesta	**Mapy.cz**	**Klasický**	**Optimalizovaný**	**k = 1.15**	**k = 1,2**
Ostrava-Karviná	34,5 km	31 412m	31 764m	31 464m	31 879m
Praha-Ostrava	318,9 km	311 168m	317 944m	315 561m	315 969m
Aš-Jablunkov	571.7 km	562 568m	569 181m	563 539m	570 964m
Č. Budějovice-Liberec	236.4 km	233 924m	235 704m	235 042m	235 656m
Brno-Hradec Králové	142.1 km	140 785m	140 785m	141 203m	147 338m

Tabulka 6-12: Porovnání hledání nejkratší cesty

Další porovnání je hledání nejrychlejších cest mezi oběma verzemi Dijkstrova algoritmu a cestami nalezenými přes mapy.cz. V tomto porovnání jde spíše o porovnání doby jízdy po nalezené cestě, než porovnání vzdáleností cest. Nicméně oba případy jsou zde uvedeny.

Porovnání nalezených vzdáleností se nachází v tabulce 6-13 a porovnání doby jízdy je uvedeno v tabulce 6-14. Pokud se podíváme na nalezené časy, tak je vidět, že ve většině případů jsou tyty nalezené časy podobné. Například při hledání cesty z Brna do Hradce Králové se tyto nalezené časy liší o pouhou minutu. Nicméně, v případě hledání cesty z Aše do Jablunkova se nalezený čas cesty liší o hodinu a půl. To může být způsobeno různým určováním výpočtu času jednotlivých algoritmů.

		Dijkster	
Cesta	**Mapy.cz**	**Klasický**	**Optimalizovaný**
Ostrava-Karviná	38,2 km	34 781m	34 781m
Praha-Ostrava	369,0 km	368 486m	376 501m
Aš-Jablunkov	672.2 km	650 618m	650 618m
Č. Budějovice-Liberec	250.5 km	250 114m	250 114m
Brno-Hradec Králové	142.7 km	141 900m	141 900m

Tabulka 6-13: Porovnání hledání nejrychlejší cesty (vzdálenost)

		Dijkster	
Cesta	**Mapy.cz**	**Klasický**	**Optimalizovaný**
Ostrava-Karviná	39m	31m	31m
Praha-Ostrava	3h 18m	3h	3h 15m
Aš-Jablunkov	7h 37m	6h 5m	6h 5m
Č. Budějovice-Liberec	2h 41m	2h 33m	2h 33m
Brno-Hradec Králové	1h 53m	1h 52m	1h 52m

Tabulka 6-14: Porovnání hledání nejrychlejší cesty (doba jízdy)

Porovnání výsledků hledání cest ukázalo, že oba naimplantované algoritmy (Dijkstrův i A*) jsou schopny vyhledat cestu stejnou, nebo lepší v porovnání s mapy.cz. Těchto výsledků je dosaženo v časech hledání nepřekračující 2,5 sekundy, což závisí na délce hledané cesty. Hlavně díky vhodným optimalizacím se podařilo dostat časy hledání na tuto úroveň.

Dijkstrův algoritmus je vhodný na hledání cest v malých množinách dat, nebo na množinách dat, které jsou optimalizované. Čas nalezení cesty je přímo úměrný počtu prohledaných dat.

Zato u algoritmu A* není velikost množiny dat závislá na době hledání. Tento algoritmus je velice dobře škálovatelný, díky možnosti nastavení konstanty vyhledávání. Proto je ideální řešení pro hledání nejkratších cest nad velkými množinami dat. Jeho nevýhoda spočívá v tom, že nedokáže hledat nejrychlejší cesty.

Co se týče hledání cest v České republice, tak je z předchozího srovnání jasné, že oba algoritmy se dají velice dobře použít, a že je zaručeno, že najdou vhodné cesty.

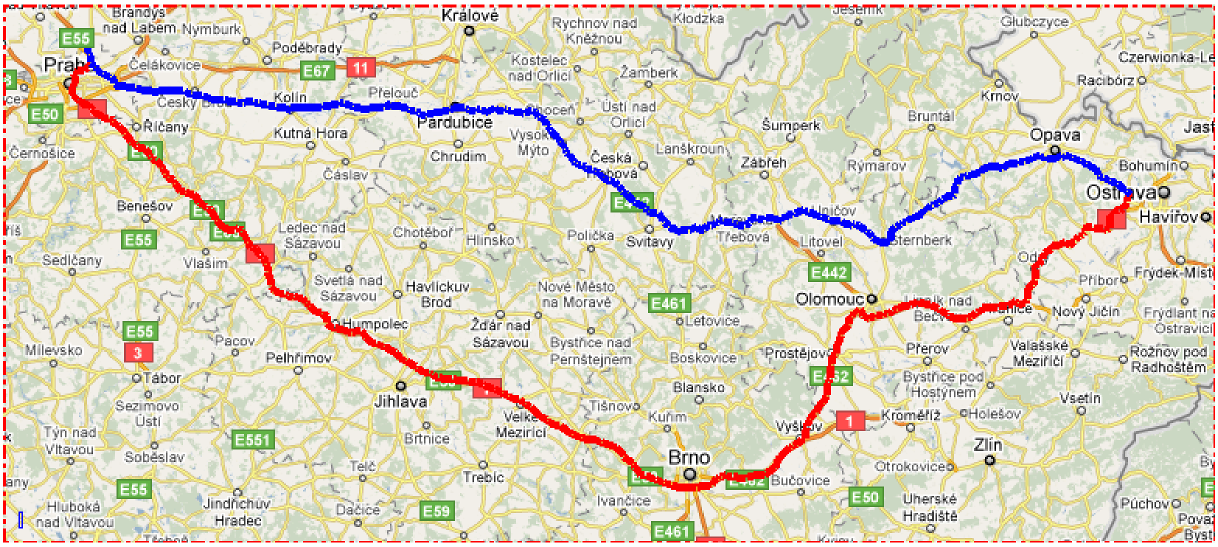
7 Vykreslení nalezené cesty

Pro vykreslení nalezené cesty do mapy byla použita open source komponenta Gmap. Tato komponenta je dostupná na adrese http://greatmaps.codeplex.com a využívá mapový podklad z maps.google.com. Její funkce spočívá v tom, že stáhne mapu podle zadaných GPS souřadnic a umožní kreslit do této mapy.

Pro správnou funkčnost s našimi naimplementovanými algoritmy ovšem bylo zapotřebí přepsat pár metod a také vstup do této komponenty. Nicméně díky tomu můžeme vykreslovat do mapy nejenom cesty, ale třeba i jednotlivé vrcholy, jak je ukázáno na obrázcích 6-3 až 6-7.

Na obrázku 7-1 je zobrazeno vykreslení nejkratší a nejrychlejší cesty z Prahy do Ostravy pomocí komponenty Gmap. Nejrychlejší cesta vede přes Brno a má červenou barvu. Nejkratší cesta má barvu modrou.

Díky vykreslení nalezené cesty můžeme vidět, kudy nalezená cesta vede a můžeme tak jednoduše porovnávat výsledky hledání jednotlivých algoritmů.



Obrázek 7-1: Vykreslení nejkratší a nejrychlejší cesty z Prahy do Ostravy

8 Závěr

V této práci jsme se zabývali algoritmy pro směrování vozidel. Měli jsme k dispozici data silniční sítě České republiky, na kterých jsme testovali vybrané algoritmy pro hledání cest a vytvářeli různé optimalizace pro urychlení běhu algoritmů.

Uvedli jsme dva algoritmy, Dijkstrův algoritmus a A* algoritmus, které jsou vhodné pro směrování vozidel v reálné silniční síti. Bylo zjištěno, že čas hledání cesty Dijkstrovým algoritmem je přímo úměrný velikosti prohledaných dat. Z tohoto důvodu bylo zapotřebí udělat optimalizace dat, nad kterými bude Disjktrův algoritmus vyhledávat. Tudíž jsme navrhli dvě optimalizace, které zrychlují hledání Dijkstrova algoritmu. První z nich je optimalizace oříznutí množiny vyhledávání, která zaručuje, že algoritmus nebude prohledávat data, která leží na opačné straně, než se nachází cílové místo. To znamená, že jsou odstraněna data, která by jinak algoritmus procházel, ale výsledná cesta by se stejně v těchto datech nenacházela. Druhou optimalizací je optimalizace vybrání vhodných dat, ta zaručuje, že se nebudou procházet ty nejhorší či nejpomalejší cesty, to záleží na tom, zda hledáme nejkratší či nejrychlejší cestu. Tyto dvě optimalizace zajišťují, vyhledání cesty Dijkstrovým algoritmem v přijatelném čase 2,3 sekundy při hledání cesty přes celou republiku z Aše do Jablunkova. Dijkstrův algoritmus dokáže vyhledat nejkratší a nejrychlejší cesty. Nicméně, vzhledem k tomu, že čas vyhledávaní závisí na množství prohledaných dat, tak je tento algoritmus spíše vhodný pro vyhledávání na malých množinách dat. Ovšem, díky provedeným optimalizacím může dobře pracovat i na velkých množinách dat.

Druhým uvedeným algoritmem je A* algoritmus. Pro tento algoritmus je klíčové to, že pracuje s odhadem vzdálenosti od aktuálního vrcholu do cílového vrcholu. Přesnost odhadu této vzdálenosti je předána do algoritmu pomocí konstanty. Ukázali jsme si, jak závisí hodnota této konstanty na čase vyhledávání a počtu prohledaných vrcholů. Zjistili jsme, že pokud má konstanta hodnotu 1, pak je chování A* algoritmu velice podobné Dijkstrovému algoritmu. Z experimentů jsme zjistili, že nejvhodnější konstanta pro hledání cest v České republice je v rozmezí 1,15 až 1,2. Hlavní výhodou tohoto algoritmu je to, že je schopen nalézt rychle cestu i ve velké množině dat, pokud má vhodně nastavenou konstantu. Nevýhoda tohoto algoritmu spočívá v tom, že nedokáže vyhledávat nejrychlejší cesty.

Porovnáním výsledků hledání cest z obou algoritmů a výsledků z webové aplikace pro hledání cest Mapy.cz jsme zjistili, že námi naimplementované a optimalizované algoritmy jsou schopny vyhledat stejnou, nebo lepší cestu ve srovnatelném čase hledání. Tímto jsme splnily požadavky stanovené na začátku práce. Tudíž lze námi vytvořené optimalizace algoritmů hodnotit jako úspěšné a vhodné pro hledání cest.

Seznam použité literatury

[1] Amit's A* Pages: *http://theory.stanford.edu/~amitp/GameProgramming/*

[2] Jakub Černý: Základní grafové algoritmy, Kapitola: Nejkratší cesta v grafu

[3] Krzysztof R. Apt: *Edsger Wybe Dijkstra (1930–2002): A Portrait of a Genius*

[4] Marcus Jenkins: Introduction to Route Calculation